Topics in Formal Languages: String Enumeration, Unary NFAs and State Complexity

by

Andrew Martinez

A thesis

presented to the University of Waterloo

in fulfilment of the

thesis requirement for the degree of

Master of Mathematics

in

Computer Science

Waterloo, Ontario, Canada, 2002

I hereby declare that I am the sole author of this thesis.

I authorize the University of Waterloo to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize the University of Waterloo to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

The University of Waterloo requires the signatures of all persons using or photocopying this thesis. Please sign below, and give address and date.

Abstract

We first consider the problems of counting the exact number of strings of a given length in a language, and of determining the ith string in lexicographical order of a given length in a language, where the language is specified by some model (usually an automaton or a grammar). We show that these problems can be solved efficiently for some models of specification, but not for others.

Next, we consider the problem of efficient computation of regular expressions from unary NFAs. We use the existence of a normal form for unary NFAs, called Chrobak normal form, to obtain an efficient conversion algorithm. The conversion algorithm was coded in the C programming language and is available for download.

We also examine ambiguous unary NFAs. We give an upper and lower bound on the length of a shortest string accepted by a unary NFA with two distinct accepting computations, and give an efficient algorithm to decide whether a unary NFA in Chrobak normal form is ambiguous or not.

Lastly, we examine the minimization operation on languages. If L is a language, then the minimization of L, denoted L_{min}, is the subset of L consisting of the lexicographically smallest strings of each length in L. We prove a lower bound on the state complexity of this operation.

Acknowledgements

First, I would like to thank my supervisor, Jeffrey Shallit, for his guidance and encouragement over the past two years. I would also like to thank Jonathan Buss and Timothy Chan for reading my thesis and offering many helpful suggestions, Therese Biedl for discussions on the UNION problem described in section 4.3, the anonymous referees of the Descriptional Complexity of Formal Systems (DCFS) 2002 workshop for comments and suggestions regarding Chapter 4, Michael Domaratzki, Keith Ellul and Ming-Wei Wang for the many valuable discussions and sharing of ideas, and last but not least, my parents for their continued love and support.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Introduction

We will be focusing on three areas in formal language theory: the areas of string enumeration (or counting), unary nondeterministic finite automata, and state complexity.

1.1.1 String Enumeration

String enumeration involves counting the exact number of strings, over some fixed alphabet, which satisfy a given property. Our main interest is counting the number of strings of a specified length in a language, where the language is specified by some model (usually an automaton or a grammar). The model of specification is important — an efficient algorithm may exist for one model of specification, but not another.

We also investigate the problem of computing the ith string in lexicographical order of a given length in a language, where again the language is specified by some model. In order for a lexicographical ordering to make sense, we define a total ordering on the underlying alphabet and extend this to a partial order on strings. Again, efficient algorithms exist for some language descriptors but not for all.

1.1.2 Unary Nondeterministic Finite Automata

A **unary nondeterministic finite automaton**, or unary NFA, is a nondeterministic finite automaton over a single-letter alphabet. It is known that for every unary NFA, there exists a unary NFA in a "normal form" which accepts the same language. A unary NFA in this "normal form" has a "tail" and several "loops" attached to the end of the tail. We examine two problems involving unary NFAs in normal form.

First, we present an efficient algorithm which converts an arbitrary unary NFA on n states to a regular expression of size $O(n^2)$. The normal form is useful because regular expressions can be easily obtained from unary NFAs in normal form. It is shown that the running time of this algorithm is polynomial in n. This algorithm was coded in the C programming language and is available for download. The actual running time of the program was compared to the expected running time based on theoretical analysis, and the results were recorded.

Second, we examine the ambiguity problem for unary NFAs. The ambiguity problem is as follows: given a unary NFA M, does there exist a string w for which M has two distinct accepting computations? If the answer is yes, M is said to be ambiguous and w is an ambiguous string for M. We consider the ambiguity problem for general unary NFAs and give upper and lower bounds on the length of the shortest ambiguous string for an ambiguous unary NFA. We also consider the ambiguity problem for inputs in normal form and give an efficient algorithm for deciding the ambiguity problem in this case.

1.1.3 State Complexity

State complexity is a descriptional complexity measure for regular languages, based on deterministic finite automata. The state complexity of a regular language L is defined to be the number of states in the smallest deterministic finite automaton accepting L. Let L be any language. L_{min}, referred to as the minimization of L, is defined to be the subset

of L consisting of the lexicographically smallest strings of each length in L. It is known that the set of regular languages is closed under minimization; in other words, if L is a regular language, then L_{min} must also be regular. This presents the following question: find upper and lower bounds on the state complexity of L_{min} in the worst case. We present a superpolynomial lower bound on the state complexity of L_{min}.

1.2 Notation and Conventions

Let $\mathbb{N} = \{0, 1, 2, \ldots\}$ denote the set of natural numbers. Our model of computation is the "naive bit complexity" model [2, p. 43]. Furthermore, we define

$$\lg n = \begin{cases} 1, & \text{if } n = 0; \\ 1 + \lfloor \log_2 |n| \rfloor, & \text{otherwise.} \end{cases}$$

This allows us to use $\lg n$ to denote the number of bits in the binary representation of the integer n, excepting the sign bit [2, p. 41]. Using the naive bit complexity model, the operation $a + b$ uses $(\lg a) + (\lg b)$ bit operations and the operation $a \times b$ uses $(\lg a) \times (\lg b)$ bit operations.

Chapter 2

Efficient Computation of $|L \cap \Sigma^n|$

2.1 Introduction

We are interested in algorithms which, given a language L (usually represented by an automaton or a grammar) and an integer $n \geq 0$ as input, compute the number of strings in L of length n as efficiently as possible. We begin with a definition:

Definition 2.1 *Given a language L and an integer n, define t_n to be the number of strings in L of length n, i.e., $t_n = |L \cap \Sigma^n|$.*

2.2 Computation of $|L \cap \Sigma^n|$ When L is Regular

First, we will examine the situation where L is a regular language. It is well known that a language is regular if and only if it is accepted by a deterministic finite automaton (DFA) $M = (Q, \Sigma, \delta, q_0, F)$, where Q is a finite nonempty set of states, Σ is a finite nonempty alphabet, $\delta : Q \times \Sigma \to Q$ is the transition function, $q_0 \in Q$ is the start state, and $F \subseteq Q$ is the set of final states [13].

Definition 2.2 *Let S be an $p \times q$ matrix. Define $S_{i,j}$ to be the element in the ith row and*

jth column of S, where $0 \leq i \leq p-1$ and $0 \leq j \leq q-1$. Similarly, if S is a $p \times p$ matrix, define $S^n_{i,j}$ to be the element in the ith row and jth column of S^n, where $0 \leq i, j \leq p-1$.

Definition 2.3 *Let $M = (\{q_0, q_1, \ldots, q_{k-1}\}, \Sigma, \delta, q_0, F)$ be a finite automaton (either deterministic or nondeterministic) with k states. Define the **adjacency matrix** of M to be the $k \times k$ matrix A where:*

$$A_{i,j} = |\{a \in \Sigma : \delta(q_i, a) = q_j\}|.$$

Note that the (i,j)th entry of A equals the number of distinct labeled transitions from state q_i to state q_j. This property extends to powers of A as follows: $A^n_{i,j}$ equals the number of distinct labeled paths of length n from state q_i to state q_j [6, Lemma 2.5, p. 11].

2.2.1 When L is Specified by a Deterministic Finite Automaton (DFA)

Consider the situation where L is regular and is specified by a DFA $M = (Q, \Sigma, \delta, q_0, F)$. The properties of the adjacency matrix give us the following simple algorithm for computing t_n:

`ALGORITHM 2.1`

(Input: DFA $M = (Q, \Sigma, \delta, q_0, F)$, integer $n \geq 0$)

(Output: $t_n = |L(M) \cap \Sigma^n|$)

`begin`

$t_n \leftarrow 0$

$A \leftarrow$ `adjacency matrix of` M

$B \leftarrow A^n$

$t_n \leftarrow \sum_{q_j \in F} B_{0,j}$

```
    return t_n
end
```

Proposition 2.1 *Given a DFA $M = (Q, \Sigma, \delta, q_0, F)$ and an integer $n \geq 0$, Algorithm 2.1 correctly computes t_n using $O(|Q|^3 n^2)$ bit operations.*

Proof. Observe that $B_{0,j} = A^n_{0,j}$ denotes the number of distinct labeled paths of length n from q_0 to q_j. Thus, summing $B_{0,j}$ over all j for which $q_j \in F$ gives the number of distinct labeled paths of length n leading from the start state to some final state. Let us denote this quantity by s_n. Since M is deterministic, there is a 1-1 relation between paths in M and strings in $L(M)$ and thus s_n is precisely equal to t_n, the desired quantity.

In terms of bit operations, computation of the adjacency matrix A uses $O(|Q|^2)$ bit operations. (Note that the entries of A are at most $|\Sigma|$, which is constant.) Since multiplication of $m \times m$ matrices can be performed using $O(m^3)$ arithmetic operations [1, Ch. 6], the total number of bit operations used is:

$$\begin{aligned} O(\sum_{i=0}^{\lg n} (|Q|^3 (\lg |\Sigma|^{2^i})^2)) &= O(|Q|^3 \sum_{i=0}^{\lg n} (2^i \lg |\Sigma|)^2) \\ &= O(|Q|^3 (\lg |\Sigma|)^2 \sum_{i=0}^{\lg n} (4^i)) \\ &= O(\frac{1}{3} |Q|^3 (\lg |\Sigma|)^2 (4^{\lg n+1} - 1)), \end{aligned}$$

which is $O(|Q|^3 n^2)$. □

Note that, although t_n can be as large as $|\Sigma|^n$, it can grow as slowly as polynomially in n. For instance, let M be the DFA shown in Figure 2.1. It is easily verified that $L(M) = \{w \in \{0,1\}^* : w \text{ contains exactly two 1's}\}$ and that $t_n = \frac{n(n-1)}{2}$. Since the entries of A^n can be as large as $|\Sigma|^n$, explicitly computing each element of A^n might require a polynomial number of bit operations in n, even though we are not interested in all of the entries of A^n. This

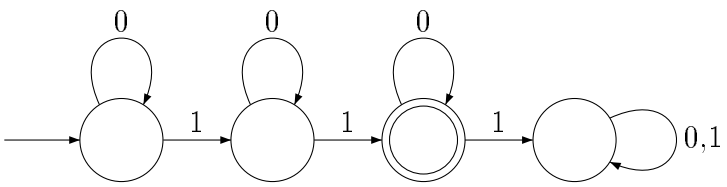

Figure 2.1: A DFA accepting strings in $\{0,1\}^*$ with exactly two 1's.

is particularly relevant when t_n is logarithmic in $|\Sigma|^n$. If we can avoid computing unneeded entries in A^n, the number of bit operations used to compute t_n will be polynomial in $|Q|$, $\lg n$ and $\lg t_n$, instead of being polynomial in $|Q|$ and n, according to Proposition 2.1.

Observe that the entries in an adjacency matrix are always non-negative. Using this fact, we can bound the size of the entries of the matrix when performing matrix multiplication, if we allow the entries to assume an additional special value, g. We define this formally, as follows:

Definition 2.4 *Define a g-**matrix** to be a matrix whose entries are elements of $\mathbb{N} \cup \{g\}$, where g is a distinguished indeterminate.*

When working with g-matrices, it is necessary also to specify a **bound**, $b \geq 0$. Informally speaking, a g-matrix S with a bound b has the property that the non-g entries of S do not exceed b. More precisely, addition and multiplication of elements in $\mathbb{N} \cup \{g\}$ is defined as follows:

Definition 2.5 *Fix a bound $b \geq 0$. Define addition (denoted by the binary operand $\oplus$) and multiplication (denoted by the binary operand $\otimes$) over the elements in $\mathbb{N} \cup \{g\}$, referred to as g-**addition** and g-**multiplication** respectively, as follows:*

- $g \oplus g = g$
- *For all $n \in \mathbb{N}$, $g \oplus n = n \oplus g = g$*

- *For all* $m, n \in \mathbb{N}$, $m \oplus n = \begin{cases} g, & \text{if } m + n > b; \\ m + n, & \text{otherwise.} \end{cases}$
- $g \otimes g = g$
- $g \otimes 0 = 0 \otimes g = 0$
- *For all* $n \geq 1$, $g \otimes n = n \otimes g = g$
- *For all* $m, n \in \mathbb{N}$, $m \otimes n = \begin{cases} g, & \text{if } m \times n > b; \\ m \times n, & \text{otherwise.} \end{cases}$

Using the above, we extend the definition to addition and multiplication of g-matrices, as follows:

Definition 2.6 *Fix a bound* $b \geq 0$. *Let* S *and* T *be* g*-matrices of dimension* $p \times q$, *and let* U *be a* g*-matrix of dimension* $q \times r$. *Define addition (denoted by the binary operand* $\oplus$*) and multiplication (denoted by the binary operand* $\otimes$*) of* g*-matrices, referred to as* g***-matrix addition*** *and* g***-matrix multiplication*** *respectively, as follows:*

- $(S \oplus T)_{i,j} = S_{i,j} \oplus T_{i,j}$
- $(T \otimes U)_{i,j} = \bigoplus_{k=1}^{q} (T_{i,k} \otimes U_{k,j})$

Example 2.1 *Consider the two* g*-matrices:*

$$S = \begin{bmatrix} 0 & 1 \\ 2 & g \end{bmatrix}, \; T = \begin{bmatrix} 1 & 1 \\ g & 0 \end{bmatrix}.$$

With a bound of $b = 2$, *we get:*

$$S \oplus T = \begin{bmatrix} 1 & 2 \\ g & g \end{bmatrix}, \; S \otimes T = \begin{bmatrix} g & 0 \\ g & 2 \end{bmatrix}.$$

Definition 2.7 *Fix a bound $b \geq 0$. Let G_b be the map which sends matrices with non-negative integer entries to g-matrices with bound b. More precisely: if S is a matrix with non-negative integer entries, then*

$$(G_b(S))_{i,j} = \begin{cases} g, & \text{if } S_{i,j} > b; \\ S_{i,j}, & \text{otherwise.} \end{cases}$$

Proposition 2.2 *Let S and T be $p \times q$ matrices and U be a $q \times r$ matrix, in which all entries are non-negative integers. Fix a bound $b \geq 0$. Then $G_b(S+T) = G_b(S) \oplus G_b(T)$ and $G_b(S \times T) = G_b(S) \otimes G_b(T)$.*

Proof. We will handle the case of addition first. Let $0 \leq i \leq p-1$ and $0 \leq j \leq q-1$ be arbitrary, and consider the value of $(S+T)_{i,j} = S_{i,j} + T_{i,j}$. It is clear that $[G_b(S+T)]_{i,j} = [G_b(S) \oplus G_b(T)]_{i,j}$ if $S_{i,j} + T_{i,j} \leq b$, so we may assume that $S_{i,j} + T_{i,j} > b$. Then $[G_b(S+T)]_{i,j} = g$ and $[G_b(S) \oplus G_b(T)]_{i,j} = [G_b(S)]_{i,j} \oplus [G_b(T)]_{i,j}$. Now, if either $[G_b(S)]_{i,j}$ or $[G_b(T)]_{i,j}$ equals g, then their sum is also g (by definition of addition in $\mathbb{N} \cup \{g\}$); if not, then $[G_b(S)]_{i,j} = S_{i,j}$ and $[G_b(T)]_{i,j} = T_{i,j}$. But $S_{i,j} + T_{i,j} > b$, so $S_{i,j} \oplus T_{i,j} = g$. Hence $[G_b(S+T)]_{i,j} = [G_b(S) \oplus G_b(T)]_{i,j}$, as required.

Now consider the case of multiplication. Let $0 \leq i \leq p-1$ and $0 \leq j \leq r-1$ be arbitrary, and consider the value of $(T \times U)_{i,j} = \sum_{k=1}^{q} (T_{i,k} \times U_{k,j})$. Again, it is clear that $[G_b(T \times U)]_{i,j} = [G_b(T) \otimes G_b(U)]_{i,j}$ if $\sum_{k=1}^{q} (T_{i,k} \times U_{k,j}) \leq b$, so let us assume that $\sum_{k=1}^{q} (T_{i,k} \times U_{k,j}) > b$. Then $[G_b(T \times U)]_{i,j} = g$ and $[G_b(T) \otimes G_b(U)]_{i,j} = \bigoplus_{k=1}^{q} ([G_b(T)]_{i,k} \otimes [G_b(U)]_{k,j})$. If any of the terms $[G_b(T)]_{i,k} \otimes [G_b(U)]_{k,j}$ equal g, then the entire sum is also g (by definition of g-addition); if not, then for all i and j, we have that $[G_b(T)]_{i,k} \otimes [G_b(U)]_{k,j} = T_{i,k} \times U_{k,j}$ (by definition of g-multiplication). But $\sum_{k=1}^{q} (T_{i,k} \times U_{k,j}) > b$, so we must have that $\bigoplus_{k=1}^{q} ([G_b(T)]_{i,k} \otimes [G_b(U)]_{k,j}) = g$, as required. □

Since the entries of adjacency matrices are always non-negative, it is possible, using g-matrices and bounds, to compute t_n without having to explicitly compute each entry of A^n (where A is the adjacency matrix) via the following method:

```
ALGORITHM 2.2
```
(Input: DFA $M = (Q, \Sigma, \delta, q_0, F)$*, integer* $n \geq 0$*)*

(Output: $t_n = |L(M) \cap \Sigma^n|$*)*

```
begin
   b ← 2
   A ← adjacency matrix of M
   loop forever
      t_n ← 0
      B ← G_b(A)
      C ← B^n (using g-matrix multiplication)
      t_n ← ⊕_{q_j ∈ F} C_{0,j}
      if t_n ≠ g then break
      b ← b × 2
   end loop
   return t_n
end
```

Proposition 2.3 *Given a DFA* $M = (Q, \Sigma, \delta, q_0, F)$ *and an integer* $n \geq 0$*, Algorithm 2.2 correctly computes* t_n *using* $O(|Q|^3 (\lg n) (\lg t_n)^3)$ *bit operations.*

Proof. We start by obtaining the adjacency matrix A in the usual manner. In each iteration of the loop, we set B to be the g-matrix equivalent of A with bound b. We then compute $C = B^n$ using g-matrix multiplication and t_n using g-addition. If the obtained value of t_n

is g, then $t_n > b$. We then double the bound and repeat the process until a non-g value is obtained for t_n. The algorithm will terminate once $b \geq t_n$.

Consider the running time of this algorithm. Since g-multiplication is being used, the largest non-g entry possible at any time is at most $2t_n$. Since at most $\lg t_n$ recomputations are required, the total number of bit operations is given by:

$$\begin{aligned} O(\sum_{i=0}^{\lg t_n} (|Q|^3 (\lg n)(\lg(2^i b))^2)) &= O(|Q|^3 (\lg n) \sum_{i=0}^{\lg t_n} ((i + \lg b)^2)) \\ &= O(|Q|^3 (\lg n) \sum_{i=0}^{\lg t_n} (i^2 + 2i \lg b + (\lg b)^2)). \end{aligned}$$

Since the sum of the first m squares is $\frac{m(m+1)(2m+1)}{6}$, the number of bit operations in the sum above is $O(|Q|^3 (\lg n)(\lg t_n)^3)$. □

Now, suppose that instead of doubling the bound in each iteration of the loop in Algorithm 2.2, we *square* it. Then the largest non-g entry possible at any time is at most t_n^2. Since at most $\lg \lg t_n$ recomputations are required, the total number of bit operations used is:

$$\begin{aligned} O(\sum_{i=0}^{\lg \lg t_n} (|Q|^3 (\lg n)(\lg b^{2^i})^2)) &= O(|Q|^3 (\lg n) \sum_{i=0}^{\lg \lg t_n} ((2^i \lg b)^2)) \\ &= O(|Q|^3 (\lg n)(\lg b)^2 \sum_{i=0}^{\lg \lg t_n} (4^i)) \\ &= O(\frac{1}{3} |Q|^3 (\lg n)(\lg b)^2 (4^{\lg \lg t_n + 1} - 1)). \end{aligned}$$

This sum is $O(|Q|^3 (\lg n)(\lg t_n))$, which gives us the following improvement over Proposition 2.3:

Theorem 2.1 *Given a DFA $M = (Q, \Sigma, \delta, q_0, F)$ and an integer $n \geq 0$, t_n can be computed using $O(|Q|^3 (\lg n)(\lg t_n))$ bit operations.*

Note, however, that *exponentiating* the bound at each step (i.e., changing the bound from b to, say, 2^b at each step) does not yield a further improvement. Rather, exponentiation actually increases the number of bit operations required. If the bound is exponentiated at each step, the largest non-g entry possible at any time is at most 2^{t_n}, which means that each g-multiplication yielding A^n will use $O(|Q|^3 (\lg n)\, t_n^2)$ bit operations.

2.2.2 When L is Specified by a Nondeterministic Finite Automaton (NFA)

Consider the problem of computing t_n when L is specified by a nondeterministic finite automaton (NFA) $M = (Q, \Sigma, \delta, q_0, F)$. Gore, Jerrum, Kannan, Sweedyk and Mahaney showed [11] that the problem is #**P**-complete if $|\Sigma| \geq 2$. For the case $|\Sigma| = 1$, we present two efficient algorithms for computing t_n. The first algorithm uses g-matrices and is as follows:

```
ALGORITHM 2.3
```
(Input: unary NFA $M = (Q, \{a\}, \delta, q_0, F)$*, integer* $n \geq 0$*)*

(Output: $t_n = |L(M) \cap \Sigma^n|$*)*

```
begin
   b ← 0
   A ← adjacency matrix of M
   B ← G_b(A)
   C ← B^n (using g-matrix multiplication)
   if ⊕_{q_j ∈ F} C_{0,j} = g then
      return 1
   else
      return 0
```

```
   end if
end
```

Theorem 2.2 *Given a unary NFA $M = (Q, \Sigma, \delta, q_0, F)$ and an integer $n \geq 0$, Algorithm 2.3 correctly computes t_n using $O(|Q|^3 \lg n)$ bit operations.*

Proof. Since $|\Sigma| = 1$, $t_n \in \{0, 1\}$ for all n and hence the problem of computing t_n is reduced to deciding if $a^n \in L(M)$. This is done using the standard adjacency matrix technique.

As for the running time of this algorithm, observe that with a bound of $b = 0$, the size of the numbers encountered in g-matrix multiplication is not an issue. Thus, modifying the analysis used in the deterministic case shows that the above algorithm for computing t_n uses $O(|Q|^3 \lg n)$ bit operations. □

A second algorithm uses two bit arrays to simulate execution of the unary NFA on its input, and is as follows. In this algorithm, $Q = \{q_0, q_1, \ldots, q_{|Q|-1}\}$ and $\Sigma = \{a\}$.

```
ALGORITHM 2.4
(Input: unary NFA M = (Q, {a}, δ, q_0, F), integer n ≥ 0)
(Output: t_n = |L(M) ∩ Σ^n|)
(Note: OLDARRAY and NEWARRAY are bit arrays of size |Q|.)
begin
   for j = 0 to |Q| − 1 do
      OLDARRAY[j] ← 0
      NEWARRAY[j] ← 0
   end for
   NEWARRAY[0] ← 1
   for i = 1 to n do
```

```
      for j = 0 to |Q| − 1 do
         OLDARRAY[j] ← NEWARRAY[j]
         NEWARRAY[j] ← 0
      end for
      for j = 0 to |Q| − 1 do
         if OLDARRAY[j] = 1 then
            for k = 0 to |Q| − 1 do
               if q_k ∈ δ(q_j, a) then
                  NEWARRAY[k] ← 1
               end if
            end for
         end if
      end for
   end for
   for i = 0 to |Q| − 1 do
      if NEWARRAY[i] = 1 and q_i ∈ F then
         return 1
      end if
   end for
   return 0
end
```

Theorem 2.3 *Given a unary NFA $M = (Q, \{a\}, \delta, q_0, F)$ and an integer $n \geq 0$, Algorithm 2.4 correctly computes t_n using $O(|Q|^2\, n)$ bit operations.*

Proof. As mentioned in the proof of Theorem 2.2, computing t_n can be reduced to deciding whether $a^n \in L(M)$. NEWARRAY is used to record the subset of states that M is in when

processing a^n. At the beginning of the algorithm, `NEWARRAY` consists only of the start state. In each iteration of the outer loop, `NEWARRAY` is replaced by the set of states which are reachable in one move from `OLDARRAY`, thus simulating the behaviour of M on input a^n. Thus, after n iterations, `NEWARRAY` consists of states which are reachable from q_0 in n moves. The last `for` loop checks if any of these states are accepting states.

In terms of bit operations, observe that the outermost `for` loop iterates n times, while there are two inner `for` loops, one nested inside the other, which each iterate $|Q|$ times. □

2.2.3 When L is Specified by an Unambiguous Nondeterministic Finite Automaton (UFA)

Recall that an **unambiguous finite automaton** (UFA) is an NFA in which there is a unique accepting computation for every accepted string [14]. Observe that Algorithm 2.2 for DFAs also works for UFAs because the one-to-one relationship between accepting paths in M of length n and strings in $L(M)$ of length n holds for UFAs. Thus, Theorem 2.1 applies and t_n can be computed efficiently if L is specified by a UFA.

2.3 Computation of $|L \cap \Sigma^n|$ When L is Context-Free

We will now examine the case where L is specified by a context-free grammar (CFG) G. Here, the issue which determines whether or not t_n can be computed efficiently is whether G is ambiguous or not. Recall that a context-free grammar G is **ambiguous** if there exist two distinct parse trees for some string $w \in L(G)$. Also, a context-free grammar $G = (V, \Sigma, P, S)$ is said to be in **Chomsky normal form** (CNF) if every production in G is of the form $A \rightarrow BC$ or $A \rightarrow a$, where $A, B, C \in V$ and $a \in \Sigma$. The CNF conversion algorithm by Hopcroft and Ullman [13, p.92] proceeds in three steps: first, nullable productions are

removed; second, unit productions are removed; and last, long productions are converted to short ones. Unfortunately, this algorithm allows an exponential increase in the number of productions. If, however, the third step is performed before the first two steps, a CNF conversion algorithm is obtained which uses $O(k\,|P|^2)$ bit operations, where k is the length of the longest production in P and $|P|$ is the number of productions in G.

2.3.1 When L is Specified by an Unambiguous Context-Free Grammar

The problem of computing t_n has an efficient algorithm if the context-free language (CFL) L is specified by an unambiguous CFG. Mairson gives an algorithm [18] to compute $|L(G) \cap \Sigma^n|$, given an unambiguous CFG G in CNF, using dynamic programming. This algorithm actually counts the number of *derivations* from the start symbol producing strings of length n, making use of the fact that there is a one-to-one correspondence between the number of derivations of strings of length n and the actual number of strings of length n in when G is unambiguous. On input $G = (V, \Sigma, P, S)$ in CNF, Mairson's algorithm uses $O(n^2\,|P|\,|V|\,\lg t_n)$ bit operations and a table of size $n\,|V|$ for storage. Thus, if we take in account the cost of converting the input to CNF, Mairson's algorithm uses a total of $O(kn^2\,|P|^2\,|V|\,\lg t_n)$ bit operations.

2.3.2 When L is Specified by an Ambiguous Context-Free Grammar

Recall that the class of regular grammars consists of all left-linear and right-linear grammars. A right-linear grammar is a CFG in which all of the productions have the form $A \to wB$ or $A \to w$, where A and B are nonterminals and $w \in \Sigma^*$; a left-linear grammar is a CFG in which the productions are of the form $A \to Bw$ or $A \to w$.

If G is an ambiguous CFG, then computing $t_n = |L(G) \cap \Sigma^n|$ is likely to be difficult.

Gore, Jerrum, Kannan, Sweedyk and Mahaney showed [11] that the problem of computing $|L(G) \cap \Sigma^n|$ is **#P**-complete even when G is restricted to be a regular grammar. Note that regular grammars are not inherently ambiguous; one can convert a given ambiguous regular grammar to an equivalent unambiguous one. However, the resulting grammar could be exponentially larger than the original one.

2.4 Other Work in the Area

Since computing $t_n = |L(G) \cap \Sigma^n|$ is likely to be difficult when G is ambiguous, other approaches have been investigated.

Gore, Jerrum, Kannan, Sweedyk and Mahaney [11] consider the problem of efficient estimation of t_n. It turns out that this problem is computationally equivalent to the problem of almost uniform sampling of strings in $L(G) \cap \Sigma^n$. They show that the complexity of almost random sampling is not constrained by the **#P**-completeness of computing t_n exactly, and present an algorithm which performs this task with time complexity $\epsilon^{-2}(n|G|)^{O(\lg n)}$, where $|G|$ is a measure of the size of the grammar G and ϵ bounds the variation of the output string.

Bertoni, Goldwurm and Sabadini [4] have considered the problem of computing the number of strings of given length in CFLs from a computational complexity point of view. They show that for unambiguous CFLs, the problem of computing $|L(G) \cap \Sigma^n|$ belongs to the complexity class $\mathbf{NC}_2$ [22]. On the other hand, they show that there exists an inherently ambiguous CFL L such that if $|L \cap \Sigma^n|$ were computable in polynomial time, then $\mathbf{E} = \mathbf{NE}$, where $\mathbf{E}$ and $\mathbf{NE}$ are the complexity classes $\cup_{c \geq 1} \mathbf{TIME}(2^{cn})$ and $\cup_{c \geq 1} \mathbf{NTIME}(2^{cn})$, respectively.

Litow [17] considers a slightly different problem: he presents efficient algorithms for computing the number of distinct leftmost derivations of length n strings in $L(G)$, given a

CFG G and an integer $n \geq 0$. This quantity equals the number of strings of length n in $L(G)$ only when G is unambiguous; when G is ambiguous, the number of distinct leftmost derivations exceeds $|L(G) \cap \Sigma^n|$, as strings generated by an ambiguous CFG may have two or more distinct leftmost derivations.

Chapter 3

Computing the ith Word of Length n in a Language

3.1 Introduction

We now turn to the following problem: given a language L and integers $n \geq 0, i \geq 1$, compute the ith word in lexicographical order of length n in L. In order for a lexicographical ordering on strings to make sense, we first need to define a total order on Σ.

Definition 3.1 *Let $\Sigma = \{x_1, x_2, \ldots, x_s\}$. Define a binary relation $\prec$ on $\Sigma \times \Sigma$ as follows: $x_i \prec x_j$ if and only if $i < j$. We write $x_i \preceq x_j$ if $x_i = x_j$ or $x_i \prec x_j$.*

We now explain formally what it means for a word to precede another in lexicographical order, by extending $\prec$ to a partial order on $\Sigma^* \times \Sigma^*$.

Definition 3.2 *Let $y = y_1 y_2 \ldots y_m$ and $z = z_1 z_2 \ldots z_p$ be elements of Σ^*, where each $y_i, z_j \in \Sigma$ and $m, p \geq 1$. We say $y \prec z$ if $m = p$ and either $y_1 \prec z_1$ or there exists some $k, 2 \leq k \leq m$, such that $y_i = z_i$ for all $1 \leq i < k$ but $y_k \prec z_k$. We write $y \preceq z$ if either $y = z$ or $y \prec z$.*

Note that y and z are incomparable under the relation $\prec$ when $|y| \neq |z|$. We will fix Σ and use the lexicographical ordering defined by $\prec$ for the rest of this chapter.

3.2 Computing the ith Word of Length n When L is Regular

3.2.1 When L is Specified by a DFA

Consider the case where L is specified by a DFA M. Since t_n can be computed efficiently from M and n by Theorem 2.1, we can use it to obtain an efficient algorithm computing the ith word of $L \cap \Sigma^n$, as follows:

```
ALGORITHM 3.1
```
(Input: DFA $M = (Q, \Sigma, \delta, q_0, F)$*, integer* $n \geq 0$*, integer* $i \geq 1$*)*

(Output: w*, the* i*th word in lexicographical order of length* n *in* $L(M)$*)*

```
begin
```
$t_n \leftarrow |L(M) \cap \Sigma^n|$

`if` $i > t_n$ `then return FAIL`

$w \leftarrow \epsilon$

`for` $j = 1$ `to` n `do`

  `for` $k = 1$ `to` s `do`

    $t_{n-j,k} \leftarrow |L(M') \cap \Sigma^{n-j}|$ *(where* $M' = (Q, \Sigma, \delta, \delta(q_0, wx_k), F)$*)*

    `if` $t_{n-j,k} \geq i$ `then`

      $w \leftarrow wx_k$

      `break`

    `else`

```
            i ← i − t_{n−j,k}
         end if
      end for
   end for
   return w
end
```

Theorem 3.1 *Given a DFA $M = (Q, \Sigma, \delta, q_0, F)$, an integer $n \geq 0$ and an integer $i \geq 1$, Algorithm 3.1 correctly computes the ith word of length n in $L(M)$ using $O(|Q|^3\, n\, (\lg n)\, (\lg t_n))$ bit operations.*

Proof. First, observe that it does not make sense to ask for the ith word of a sequence if the sequence contains less than i strings in total. Thus, it must be verified that $t_n \geq i$ before proceeding.

Consider the first iteration of this algorithm. Note that $t_{n-1,k}$ represents the number of strings in $L(M)$, of length n, whose first letter is x_k (recall that $\Sigma = \{x_1, x_2, \ldots, x_s\}$). To see why this is the case, observe that the DFA M' is identical to M except for the start state, and hence for any word y we have that $y \in L(M') \cap \Sigma^{n-1}$ if and only if $x_k y \in L(M) \cap \Sigma^n$. We then compare $t_{n-1,k}$ to i. If $t_{n-1,k} \geq i$, then the ith word of length n in $L(M)$ must start with the letter x_k. Now we repeat the algorithm on the rest of the word, asking for the ith word of length $n-1$ generated by the DFA with initial state $\delta(q_0, x_k)$. On the other hand, if $t_{n-1,k} < i$, then the first letter cannot be x_k. However, we can now *subtract* $t_{n-1,k}$ from i, reducing the problem to finding the $(i - t_{n-1,k})$th word of length n in $L(M)$, as the subtraction allows us to ignore the first $t_{n-1,k}$ strings of length n in $L(M)$. This process must eventually terminate because $t_n \geq i$. Once the first letter of w has been found (suppose it is x_m), we then repeat the procedure on the next letter, using the DFA identical to M

except with start state $\delta(q_0, x_m)$. Thus, after n iterations of the outer loop, we will have the complete word.

Now consider the complexity of this algorithm. Since t_n can be computed using $O(|Q|^3 (\lg n)(\lg t_n))$ bit operations by Theorem 2.1, the total number of bit operations used is:

$$\begin{aligned} O(\sum_{j=1}^{n} (|Q|^3 (\lg(n-j))(\lg t_n))) &= O(|Q|^3 (\lg t_n)(\sum_{j=1}^{n-1} \lg j)) \\ &= O(|Q|^3 (\lg t_n) \lg((n-1)!)) \\ &= O(|Q|^3 n (\lg n)(\lg t_n)). \end{aligned}$$

(Note that the size of the alphabet, s, is considered a constant.) □

3.2.2 When L is Specified by an NFA

Since the problem of computing t_n is #**P**-complete when L is specified by an NFA over a binary alphabet [11], we have the following result:

Theorem 3.2 *Given an NFA $M = (Q, \Sigma, \delta, q_0, F)$, the problem of computing the ith string of length n in $L(M)$ is #**P**-hard when $|\Sigma| \geq 2$.*

Proof. We will show that the problem of computing t_n can be reduced to this problem in polynomial time. The reduction is as follows: suppose that an efficient algorithm exists for computing the ith word of length n in $L(M)$ for an NFA M, and that the algorithm outputs an error message if $i > t_n$. One could then use a binary search to determine the value of i such that the ith word of length n in $L(M)$ exists, but an error message is output when computing the $(i+1)$st word of length n in $L(M)$. This value of i is precisely t_n. Since $0 \leq t_n \leq |\Sigma|^n$, the initial value for i would be chosen to be $|\Sigma|^n/2$. As binary search is being

used, t_n would be found after at most $\lg(|\Sigma|^n) + 1$ or $O(n)$ iterations, which shows that the reduction occurs in polynomial time. □

Note that in the unary alphabet case, the problem of computing the ith string of length n in $L(M)$ for a given NFA M is trivial as there is at most one word of each length in $L(M)$.

3.3 Computing the ith Word of Length n When L is Context-Free

3.3.1 When L is Specified by an Unambiguous CFG

Given an unambiguous context-free grammar, $G = (V, \Sigma, P, S)$, it is conjectured that an efficient algorithm for computing the ith string of length n in G exists. Assuming that G is in CNF, we can use Mairson's dynamic programming algorithm [18] to compute $||A||_i$ for all $A \in V$ and all $1 \leq i \leq n$, where $||A||_i$ is defined to be $|\{w \in \Sigma^* : A \Rightarrow^* w, |w| = i\}|$, i.e., the number of strings of length i derivable from A. From here, one can use dynamic programming to compute, given a string w and an integer j, the number of strings in $L(G)$ of length j which have w as a prefix. This should give an algorithm which uses a polynomial number of bit operations in n and $|V|$.

3.3.2 When L is Specified by an Ambiguous CFG

Since the problem of computing t_n is **#P**-complete when L is specified by an ambiguous CFG G [11], we have the following result:

Theorem 3.3 *Given an ambiguous CFG $G = (V, \Sigma, P, S)$, the problem of computing the ith string of length n in $L(G)$ is **#P**-hard when $|\Sigma| \geq 2$.*

Proof. The proof of this result is identical to the proof of a similar result for NFAs given in Theorem 3.2. □

3.4 Other Work in the Area

The problem of generating random strings in $L \cap \Sigma^n$ has received much attention. Hickey and Cohen [12] presented two algorithms which, given an unambiguous CFG G with r nonterminals and an integer $n \geq 0$, generate a random string in $L(G) \cap \Sigma^n$. The first algorithm uses linear time to generate each string, but the algorithm must precompute a table of size $O(n^{r+1})$. The second algorithm requires only $O(n^2 \lg n)$ time and $O(n)$ space for precomputation, but uses $O(n^2 (\lg n)^2)$ time to generate each string.

Mairson [18] improved on the results obtained by Hickey and Cohen. He has two algorithms which generate a random string in $L(G) \cap \Sigma^n$: one uses $O(n)$ time and $O(n^2)$ space; the other uses $O(n^2)$ time and $O(n)$ space. However, his analyses do not take the size of the grammar into account.

Bertoni, Goldwurm and Santini [5] consider the problem of generating, uniformly at random, strings of length n from a finitely ambiguous context-free language. Their algorithm uses a probabilistic random access machine (RAM) and runs in $O(n^2 \log n)$ time, assuming a logarithmic cost criterion [1]. (A context-free grammar $G = (V, \Sigma, P, S)$ is finitely ambiguous if there exists $k \in \mathbb{N}$ such that for all $x \in L(G)$, the number of distinct leftmost derivations of x in G is at most k; a context-free language L is finitely ambiguous if there exists a finitely ambiguous CFG G such that $L = L(G)$.)

Gore, Jerrum, Kannan, Sweedyk and Mahaney [11] consider the problem of sampling a string almost uniformly from $L(G) \cap \Sigma^n$. See section 2.4 for more details on their work.

Mäkinen [19] investigated the problem of outputting the strings of $L(G) \cap \Sigma^n$ in lexicographical order. He gives an algorithm for generating the next word in lexicographical

order in $L(G) \cap \Sigma^n$, given the previous word, in time $O(n)$, where G is a regular grammar. (He has a similar algorithm for unambiguous CFGs; however, the algorithm requires that the grammar satisfy some additional hypotheses.) He also gives a separate algorithm for generating the first word in lexicographical order in $L(G) \cap \Sigma^n$ in time $O(n^2)$ and in space $O(n)$ for a (possibly ambiguous) CFG G. However, Mäkinen does not take the size of the grammar into account in his analysis.

Dömösi's work [10] is similar to that of Mäkinen. He gives a different algorithm to output the strings of $L(G) \cap \Sigma^n$ in lexicographical order, where G is either a regular grammar or a context-free grammar. The context-free version of his algorithm is a modification of the well-known CYK algorithm [15, 29]. He also gives an algorithm to compute, given a regular grammar or context-free grammar G and an integer $n \geq 0$, an encoding of the lexicographically first word of length n in $L(G)$ in $O(1)$ time by first obtaining the *minimization* of $L(G)$ (for a description of minimization, see Chapter 6). However, Dömösi does not take the size of the grammar into account in his analysis.

Chapter 4

Computing Regular Expressions from Unary NFAs

4.1 Introduction

We are interested in converting unary NFAs to regular expressions. Software packages such as Grail [24] can take a long time to perform the conversion; furthermore, the regular expression obtained can be very large. In this chapter, we present an algorithm which, given a unary NFA M with n states, produces in polynomial time a regular expression r specifying $L(M)$ such that the size of r is $O(n^2)$. This is done by converting the given unary NFA to a normal form, known as *Chrobak normal form*. We then show how to obtain short regular expressions from unary NFA in this form.

Chrobak showed that for every unary NFA M, there exists a unary NFA M' in Chrobak normal form accepting the same language, but an explicit algorithm performing this construction was not given. We will present an algorithm which, given an arbitrary unary NFA as input, outputs an equivalent unary NFA in Chrobak normal form in time polynomial in the number of states in the input. Recall that an NFA $M = (Q, \Sigma, \delta, q_0, F)$ is **unary** if

$|\Sigma| = 1$. Since we will only be considering unary NFAs here, we fix $\Sigma = \{a\}$ for the rest of this chapter.

Since all transitions in a unary NFA are labeled with the same letter, we can consider unary NFAs as directed graphs. We say that a directed graph $G = (V, E)$ is **strongly connected** if for all $u, v \in V$, there exists a nonempty directed path from u to v. A **strongly connected component** (SCC) of a directed graph $G = (V, E)$ is a maximal set of vertices $U \subseteq V$ such that the subgraph of G induced by U is strongly connected. Note that our definition of strong connectivity differs slightly from the definition found in [9]; in our definition, it is possible for vertices to not belong to any SCC. For instance, a digraph with one vertex and no edges is not strongly connected according to our definition, but is strongly connected according to [9].

Lastly, we define the **size** of a regular expression to be the number of symbols in it, including parentheses. For example, the size of $a + aa(aaa)^*$ is 10.

4.2 Definition of Chrobak Normal Form (ChrNF)

First we will define what it means for a unary NFA to be in Chrobak normal form. We use Chrobak's notation [8], but with one small difference:

Definition 4.1 *A unary NFA $M = (Q, \{a\}, \delta, q_0, F)$ is in **Chrobak normal form** (ChrNF) if it has the following properties:*

(a) $Q = \{q_0, \ldots, q_{m-1}\} \cup C_1 \cup \cdots \cup C_k$, where $C_i = \{p_{i,0}, p_{i,1}, \ldots, p_{i,y_i-1}\}$, for $i = 1, \ldots, k$;

(b) $\delta(q_i, a) = \{q_{i+1}\}$, for $0 \leq i < m - 1$;

(c) $\delta(q_{m-1}, a) = \bigcup_{i=1}^{k} \{p_{i,0}\}$; and

(d) $\delta(p_{i,j}, a) = \{p_{i,(j+1) \bmod y_i}\}$, for $1 \leq i \leq k$ and $0 \leq j \leq y_i - 1$.

The difference is that Chrobak defines Q to be $\{q_0, \ldots, q_m\} \cup C_1 \cup \ldots \cup C_k$; that is, he defines the initial portion of the NFA (also known as the "tail") to have $m+1$ states. But if the "tail" has $m+1$ states, then the states $\{p_{1,0}, p_{2,0}, \ldots, p_{k,0}\}$ are $m+1$ transitions away from q_0, not m, and thus the determination of final state status as described in his paper [8, Lemma 4.3] is incorrect. This problem is remedied by defining the "tail" to have m states, instead of $m+1$.

Note that a unary NFA $M = (Q, \{a\}, \delta, q_0, F)$ in ChrNF can be specified completely by $k+2$ integers $(m, k, y_1, y_2, \ldots, y_k)$, referred to as the **parameters** of M, and a set F of final states. For example, the unary NFA in Figure 4.1 is in ChrNF with $m = 3, k = 3, y_1 = 3, y_2 = 1, y_3 = 4$ and $F = \{q_2, p_{1,2}, p_{2,0}, p_{3,2}, p_{3,3}\}$. (Since each transition in a unary NFA occurs on the same input symbol, we omit labeling the transitions in diagrams of unary NFAs.)

4.3 An Efficient Algorithm to Convert Unary NFAs to Chrobak Normal Form

Chrobak [8] showed that for every unary NFA M on n states, there exists a unary NFA M' in ChrNF on $O(n^2)$ states which accepts the same language. We will present an algorithm which converts an arbitrary unary NFA to ChrNF, and analyze its running time.

This algorithm will use three subroutines. One subroutine computes the strongly connected components of a directed graph $G = (V, E)$. The algorithm described in Cormen, Leiserson and Rivest [9, Ch. 23, Sec. 5], attributed to R. Kosaraju (unpublished) and M. Sharir [26], uses $\Theta(|V| + |E|)$ bit operations. This algorithm can be modified easily to accommodate our definition of strong connectivity. The second subroutine takes as input at most n positive integers less than or equal to n, and returns the greatest common divisor

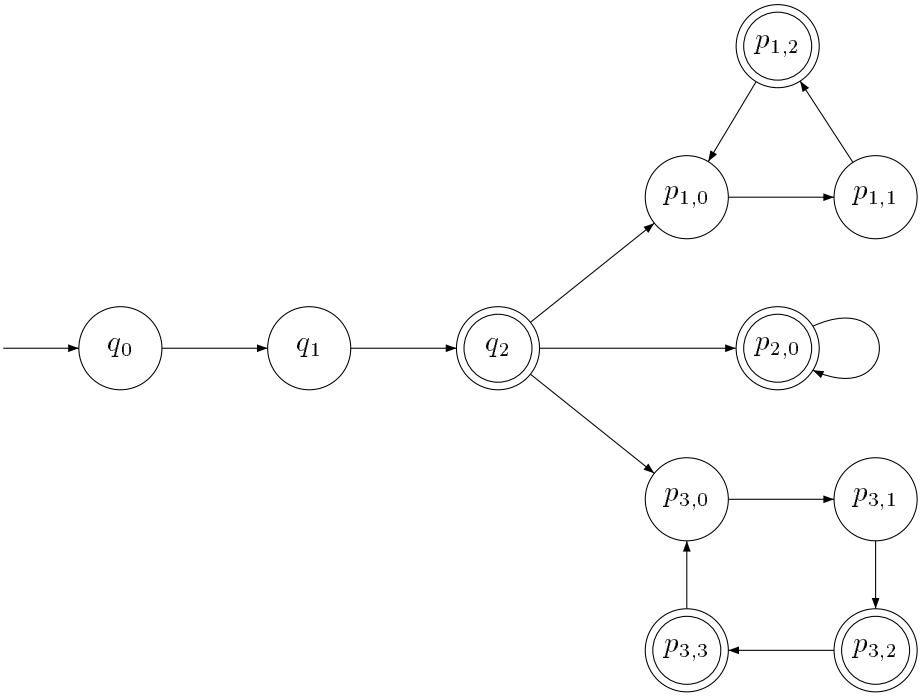

Figure 4.1: A unary NFA in ChrNF with $m = 3, k = 3, y_1 = 3, y_2 = 1$ and $y_3 = 4$.

(GCD) of the integers. Since computing the GCD of m and n can be performed using $O((\lg m)(\lg n))$ bit operations [2, p. 70] and $\sum_{i=1}^{n-1}[(\lg i)(\lg(i+1))] \leq n(\lg n)^2$, this subroutine uses $O(n(\lg n)^2)$ bit operations. The third subroutine computes the GCD of the lengths of all simple cycles in a strongly connected graph, using the second subroutine as a subroutine. We will present two algorithms here. The first algorithm uses adjacency matrices and is as follows:

ALGORITHM 4.1

(Input: a strongly connected directed graph $G = (V, E)$)

(Output: the GCD of the lengths of all simple cycles in G)

begin

```
   S ← ∅
   A ← adjacency matrix of G
   for i = 1 to |V| do
      compute A^i using Boolean matrix multiplication
      if A^i has a non-zero entry on its diagonal then
         S ← S ∪ {i}
      end if
   end for
   g ← GCD of elements in S (S ≠ ∅ as G is strongly connected)
   return g
end
```

Proposition 4.1 *Algorithm 4.1 correctly computes the GCD of the lengths of all simple cycles in the input graph G using $O(n^4)$ bit operations, where n is the number of vertices in the input graph.*

Proof. Since A is an adjacency matrix, $A^i_{j,j} \neq 0$ implies that there exists a directed path of length i from vertex j to itself in G. Thus, the directed path is either a simple cycle, or a compound cycle. (The output of the algorithm is unaffected if the directed path happens to be a compound cycle, since $\gcd(m,n) = \gcd(m,n,m+n)$ for all $m,n \geq 1$.) Iterating $|V|$ times ensures that all simple cycles in G are found. It is also clear that this algorithm terminates. Lastly, since Boolean multiplication of two $n \times n$ matrices can be performed using $O(n^3)$ bit operations, this algorithm uses $O(n^4)$ bit operations, due to the added `for` loop which iterates n times. □

The second algorithm uses adjacency lists, defined by Cormen, Leiserson and Rivest [9, p. 465] as follows:

Definition 4.2 *An **adjacency list** of a directed graph $G = (V, E)$ consists of an array* Adj *of $|V|$ lists, one for each vertex in V. For each $v \in V$, the adjacency list* Adj(u) *contains all the vertices v such that there is an edge $(u, v) \in E$.*

The adjacency list-based algorithm is as follows:

```
ALGORITHM 4.2
(Input: a strongly connected directed graph G = (V, E))
(Output: the GCD of the lengths of all simple cycles in G )
(Note: oldneighbours and newneighbours are auxiliary adjacency lists.)
begin
   S ← ∅
   Adj ← adjacency list representation of G
   for v ∈ V do
      if v ∈ Adj(v) then
         return 1 (this implies that the GCD will also be 1)
      end if
   end for
   for i = 2 to |V| do
      for v ∈ V do
         oldneighbours(v) ← Adj(v)
         newneighbours(v) ← ∅
         for w ∈ oldneighbours(v) do
            newneighbours(v) ← newneighbours(v) ∪ Adj(w)
         end for
      end for
```

```
      for v ∈ V do
         Adj(v) ← newneighbours(v)
         if v ∈ Adj(v) then
            S ← S ∪ {i} (there is a directed path from v to v of length i in G)
         end if
      end for
   end for
   g ← GCD of elements in S (S ≠ ∅ as G is strongly connected)
   return g
end
```

Let UNION(n) be the problem of computing $X \cup Y$, where X and Y are sets and $|X \cup Y|$ is at most n, and let $T(n)$ be the cost (in bit operations) of solving UNION(n). We have the following proposition:

Proposition 4.2 *Given a directed graph $G = (V, E)$, Algorithm 4.2 correctly computes the GCD of the lengths of all simple cycles in G using $O(|V| \times |E| \times T(|V|))$ bit operations.*

Proof. In Algorithm 4.2, the first `for` loop checks if there are any edges of the form $(v, v) \in E$ (i.e., simple cycles of length 1); if yes, then the GCD of all simple cycle lengths will also be 1. Otherwise, the algorithm iterates $|V|$ times, updating the adjacency list each time. Since $v \in Adj(v)$ means that there exists a directed path from v to v in G, this algorithm correctly finds all simple cycles (and all compound cycles of length at most $|V|$) in G. Thus, this algorithm works as intended.

As for the running time, observe that the two `for` loops (iterating over V and the neighbours of each vertex in V) iterate precisely $|E|$ times in total. $T(|V|)$ is the cost of computing the union of `newneighbours`(v) and $Adj(w)$. Thus, the algorithm uses a total of

$O(|V| \times |E| \times T(|V|))$ bit operations. □

Since the obvious algorithm for UNION(n) uses $O(n)$ steps, $T(n) \in O(n)$ and hence Algorithm 4.2 uses $O(n^4)$ steps in the worst case. Thus, Algorithms 4.1 and 4.2 have the same worst-case performance, but Algorithm 4.2 performs outperforms Algorithm 4.1 when G has few edges.

Before presenting the main algorithm for conversion to ChrNF, we give the following definition:

Definition 4.3 *Let $G = (V, E)$ be a directed graph, $T \subseteq V$ be a SCC of G and $P = \{p_0, p_1, \ldots, p_m\}$ be a directed path in G. P **passes through** T **last** if there exists an integer j, $0 \leq j \leq m$, such that $p_j \in T$ and none of the vertices $\{p_{j+1}, p_{j+2}, \ldots, p_m\}$ belong to any SCC of G.*

In order to execute Chrobak's algorithm on input M, we need to be able to determine, given an state $q \in Q$ and an integer $i \geq 0$, the SCCs T in M for which there exist a directed path from q_0 to q of length i passing through T last. This is done by maintaining a set $S_{q,i}$ of SCCs for each state q, called the **SCCs leading to q on input a^i**. (Note that it is possible for $S_{q,i}$ to be empty.) Clearly, if T is a SCC and $q \in T$ then $S_{q,i} = \{T\}$ if there exists a directed path of length i from q_0 to q. The interesting case occurs when q does not belong to any SCC. In this case,

$$S_{q,i} = \bigcup_{p \in Q, q \in \delta(p,a)} S_{p,i-1}.$$

The main algorithm for conversion to ChrNF is presented here:

ALGORITHM 4.3

(Input: unary NFA $M = (Q, \{a\}, \delta, q_0, F)$)

(Output: unary NFA M' where M' is in Chrobak normal form and $L(M) = L(M')$)

```
begin
   n ← |Q|
   m ← n^2 + n
   k ← number of strongly connected components of M
   T_1, T_2, ..., T_k ← the strongly connected components of M
   for i = 1 to k do
      y_i ← GCD of lengths of simple cycles in T_i
   end for
   M' ← unary NFA in Chrobak normal form with parameters m, k, y_1, y_2, ..., y_k
         with final states not yet specified
   (First determine final state status for the states in the "tail" of M')
   P_new ← {q_0}
   S_{q_0,0} ← the SCCs leading to q_0 on input ε
   for i = 0 to m − 1 do
      if P_new ∩ F ≠ ∅ then
         label q_i as a final state
      end if
      P ← P_new
      P_new ← ⋃_{p∈P} δ(p, a)
      for q ∈ P_new do
         S_{q,i+1} ← the SCCs leading to q on input a^{i+1}
      end for
   end for
   (Then determine final state status for the states in the "loops" of M')
   for j = 0 to (max_{1≤i≤k} y_i) − 1 do
      for i = 1 to k do
```

```
        if j ≥ y_i continue
        for q ∈ P_new do
            if q ∈ F and T_i ∈ S_{q,m+j} then
                label p_{i,j} as a final state
            end if
        end for
    end for
    P ← P_new
    P_new ← ⋃_{p∈P} δ(p, a)
    for q ∈ P_new do
        S_{q,m+j+1} ← the SCCs leading to q on input a^{m+j+1}
    end for
  end for
  return M'
end
```

Algorithm 4.3 starts by computing the k SCCs of M. For each SCC T_i of M, the GCD of the lengths of the simple cycles in T_i, y_i, is computed. At this point, the structure of M' is determined and it remains to specify the final states of M'. The first `for` loop in the algorithm specifies the final states in the tail of M' by simulating the execution of M on input a^i, as i ranges from 0 to $m-1$, using two lists (P and P_{new}). Note that $S_{q,i}$ is updated for each currently active state in each iteration of the loop. The second `for` loop in the algorithm specifies the final states in the loops of M' by simulating the execution of M on input a^i, as i ranges from m to $m+y-1$ (where $y = \max_{1 \leq i \leq k} y_i$). To determine whether $p_{i,j}$ should be a final state, we need to determine whether there exists an accepting path of length $m+j$ in M passing through the SCC T_i last. This is true if and only if $T_i \in S_{q,m+j}$.

It is clear that the output M' of Algorithm 4.3 is a unary NFA in ChrNF. Furthermore, $L(M') = L(M)$ [8]. Lastly, we present the following result:

Theorem 4.1 *Let M be a unary NFA with n states and k strongly connected components (when considered as a directed graph). Then Algorithm 4.3 uses $O(kn^4)$ bit operations on input M.*

Proof. Let n_i denote the number of states in the strongly connected component T_i; observe that $n_i > 0$ for each i. Observe as well that each y_i can be computed using $O(n_i^4)$ bit operations by Proposition 4.1. Since $\sum_{i=1}^{k} (n_i^p) \leq \sum_{i=1}^{k} (n^{p-1} n_i) \leq n^{p-1} \sum_{i=1}^{k} n_i \leq n^p$ for all integers $p \geq 1$, the parameters of M' (namely, m, k and the y_i) can be determined using $O(n^4)$ bit operations. The assignment of final states is done by simulating M on the appropriate input: all that is required are two arrays of size n to keep track of the previous and current states at each step of the simulation, plus a list for each state to keep track of $S_{q,i}$. (Since $S_{q,i}$ is no longer needed once $S_{q',i+1}$ has been computed for all $q' \in Q$, we need only two lists per state for storage.) Since each list has at most k elements, simulating the NFA M on input a^i can be performed using $O(ikn^2)$ bit operations. Observe that with our method, it is not necessary to simulate M on every possible input — simply simulating M on a^{m-1} is sufficient to determine final state status of all states in the tail of M'. To account for states in the loops of M', we merely simulate M on input a^{m+y-1}, where $y = \max_{1 \leq i \leq k} y_i$. Thus, the total number of steps used to determine the final states of M' is $O((m + y - 1)kn^2) = O(kn^4)$. We conclude that this algorithm uses a grand total of $O(kn^4)$ bit operations. □

4.4 Simplification of Unary NFAs in Chrobak Normal Form

Many unary NFAs in ChrNF obtained via Algorithm 4.3 can be simplified by reducing the number of states from the tail, as follows:

ALGORITHM 4.4

(Input: a unary NFA $M = (Q, \{a\}, \delta, q_0, F)$ *in ChrNF)*

(Output: a unary NFA $M' = (Q', \{a\}, \delta', q_0, F')$ *in ChrNF such that* $L(M') = L(M)$ *and* $|Q'| \leq |Q|$*)*

```
begin
   M' ← M
   repeat
      M ← M'
      P ← the set consisting of the last state in each of the loops of M
      q_{m-1} ← the last state in the tail of M
      q_{m-2} ← the second last state in the tail of M
      if (q_{m-1} ∈ F and F ∩ P ≠ ∅) or (q_{m-1} ∉ F and F ∩ P = ∅) then
         Q' ← Q − {q_{m-1}}
         F' ← F − {q_{m-1}}
         δ' ← δ
         δ'(q_{m-2}, a) ← P
      end if
   until M' = M or M' has one state in its tail
   return M'
end
```

Theorem 4.2 *Algorithm 4.4 correctly simplifies a unary NFA M in ChrNF using $O(km)$ bit operations, where k is the number of loops in M and m is the length of the tail in M.*

Proof. The simplification occurs, one state at a time, by eliminating states at the end of the tail in the given unary NFA. Define P to be the set consisting of the last state in each of the loops of M. Observe that if the last state in the tail is accepting and at least one of the states in P is accepting, then removing the last state of the tail, deleting all transitions associated with this state and adding transitions from the new last state to P yields a unary NFA in ChrNF equivalent to M, but with one fewer state in the tail. Similarly, if the last state in the tail is not accepting and none of the states in P are accepting, making the same modifications to M as described above also yields an equivalent unary NFA in ChrNF with one fewer state in the tail. This procedure is repeated on the new unary NFA until no further simplification occurs or the length of the tail is reduced to 1. As M has k loops and a tail of length m, this simplification procedure uses $O(km)$ bit operations. □

4.5 Computing Regular Expressions from Unary NFAs in Chrobak Normal Form

We now consider the following problem: given a unary NFA M' in ChrNF, find a short regular expression r such that $L(r) = L(M')$. We have the following result:

Theorem 4.3 *If M is a unary NFA with n states, a regular expression of size $O(n^2)$ specifying $L(M)$ can be found using $O(kn^4)$ bit operations, where k is the number of strongly connected components in M.*

Proof. Given M, the first step is to obtain a simplified unary NFA M' in ChrNF using Algorithms 4.3 and 4.4. By Theorem 4.1, this can be done using $O(kn^4)$ bit operations. M' will have $O(n^2)$ states. The next step is to obtain a regular expression for the tail of M'. This is done using Horner's rule to factor common subexpressions. If the tail has length $m = n^2 + n$ and the states $q_{m_1}, q_{m_2}, \ldots, q_{m_j}$ in the tail of M' are accepting states, with $0 \leq m_1 < m_2 < \cdots < m_j < m$, then Horner's rule gives the regular expression $r_t = a^{m_1}(\epsilon + a^{m_2-m_1}(\epsilon + a^{m_3-m_2}(\cdots(\epsilon + a^{m_j-m_{j-1}})\cdots)))$ for the tail of M'. Note that the size of r_t is linear in m.

The last step is to obtain a regular expression for the loops of M'. Once this has been obtained, inserting it into the spot in r_t right before the closing parentheses will yield a regular expression for $L(M)$. The regular expression for the loops of M' will have the form $(\epsilon + a^{m-m_j}(r_1 + r_2 + \cdots + r_k))$, where r_i denotes a regular expression for the ith loop. Suppose that the ith loop has length y_i and the states $p_{i,x_1}, p_{i,x_2}, \ldots, p_{i,x_{t_i}}$ in loop i are accepting, with $0 \leq x_1 < x_2 < \cdots < x_{t_i} < y_i$. Then Horner's rule gives the regular expression $r_i = a^{x_1}(\epsilon + a^{x_2-x_1}(\epsilon + a^{x_3-x_2}(\cdots(\epsilon + a^{x_{t_i}-x_{t_i-1}})\cdots)))(a^{y_i})^*$ for loop i, which has size linear in y_i. (If none of the states in loop i are accepting, set $r_i = \emptyset$.) Thus, the regular expression $(r_1 + r_2 + \cdots + r_k)$ has size $O(ky)$, where $y = \max_{1 \leq i \leq k} y_i$. Since k and y are both bounded above by n, the entire regular expression, $a^{m_1}(\epsilon + a^{m_2-m_1}(\epsilon + a^{m_3-m_2}(\cdots(\epsilon + a^{m_j-m_{j-1}}(\epsilon + a^{m-m_j}(r_1 + r_2 + \cdots + r_k)))\cdots)))$, has size $O(n^2)$. Lastly, observe that r_t can be computed using $O(m)$ bit operations and each r_i can be computed using $O(y_i)$ bit operations. Thus, the entire regular expression can be computed using $O(kn^4)$ bit operations. □

Note that simplification of regular expressions can also be performed. For instance, all regular expressions of the form $\epsilon(r)$ can be simplified to (r). This simplification was performed in the implementation of our regular expression conversion program (see section 4.7).

4.6 Some Examples

In this section, we will examine the performance of our algorithm on two unary NFAs, and compare the regular expression obtained by our algorithm to the one given by Grail on the same input.

For our first example, we consider the unary NFA M_1 depicted in Figure 4.1. Note that this unary NFA is already in ChrNF. On this input, Grail's `fmtore` command gives the regular expression $a^3a^* + a^5(a^3)^* + a^2 + a^5 + a^6(a^4)^* + a^6(a^4)^*a^3$, which has size 58. Our method computes a regular expression specifying $L(M_1)$ as follows. Using Algorithms 4.3 and 4.4, we obtain a new unary NFA in ChrNF equivalent to M_1 — let us call this unary NFA M_1' (see Figure 4.2). M_1' has one fewer state in its tail than M_1, due to simplification. Applying the method described in Theorem 4.3 to M_1' gives the regular expression $a^2((\epsilon + a^3)(a^4)^* + (a)^* + (a^3)^*)$ for $L(M_1)$. This regular expression, with size 30, is smaller than the regular expression output by Grail.

For our second example, we present a unary NFA for which our algorithm significantly outperforms Grail. Consider the unary NFA M_2 in Figure 4.3. On this input, Grail's `fmtore` command outputs a regular expression of size 63410. Using Algorithms 4.3 and 4.4, we obtain the unary NFA M_2' (see Figure 4.4). The method described in Theorem 4.3, applied to M_2', produces the regular expression $r = \epsilon + aa((a)^*)$ for $L(M_2)$. Observe that r, with size 10, is considerably smaller than the corresponding regular expression output by Grail.

4.7 A Computer Program Which Computes Regular Expressions from Unary NFAs

The methods described in Algorithms 4.3, 4.4 and Theorem 4.3 were coded in the C programming language. For the complete source code, see Appendix A. The program is also

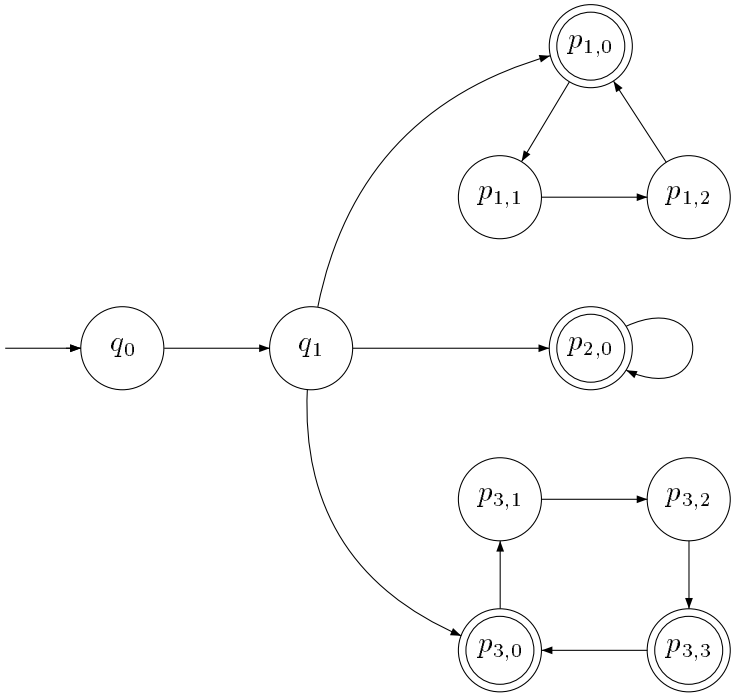

Figure 4.2: M_1': a unary NFA in ChrNF accepting $L(M_1)$. The states have been relabeled (compare to Figure 4.1).

available for download at the following URL:

`http://www.math.uwaterloo.ca/~aa2marti/chrnf.tar`

The input, an n-state unary NFA M, is read from an input file. After a unique integer from 0 to $n-1$ (inclusive) has been assigned to each state, with 0 assigned to the start state, the input file is created as follows:

n

t

$a_{1,1}\ a_{1,2}$

$a_{2,1}\ a_{2,2}$

...

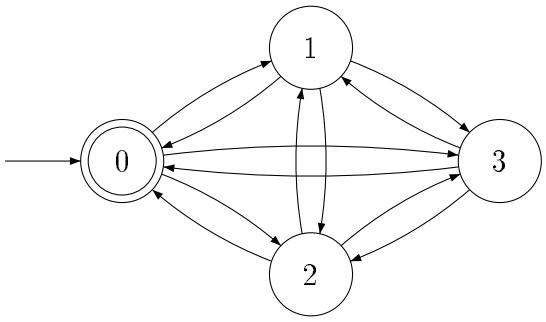

Figure 4.3: M_2: an illustration.

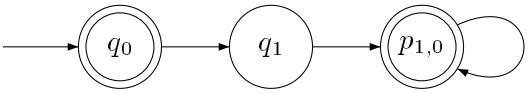

Figure 4.4: M_2': a unary NFA in ChrNF accepting $L(M_2)$.

$a_{t,1}\ a_{t,2}$

f

b_1

b_2

$\ldots$

b_f

where:

- n denotes the number of states in M;
- t denotes the number of transitions in M;
- $a_{i,1}\ a_{i,2}$ means that there is a transition from state $a_{i,1}$ to state $a_{i,2}$ in M;
- f denotes the number of accepting states in M; and
- b_i means that state b_i is an accepting state in M.

The output, a unary NFA M' in ChrNF, can either be written to the screen or redirected to a file. The format of the output file is as follows:

m

k

y_1

y_2

...

y_k

`The following states are accepting states:`

b_1

b_2

...

b_g

where:

- m denotes the number of states in the "tail" of M';
- k denotes the number of loops in M';
- y_i denotes the length of the ith cycle in M'; and
- b_i means that state b_i is an accepting state in M'.

4.7.1 Correctness and Testing

Twenty-one unary NFAs were randomly created to be used as test cases. The output from the program on these inputs was captured and compared to the expected output. These tests succeeded in finding several bugs in the program. The unary NFAs used for testing purposes had as many as 50 states and 225 transitions.

4.7.2 Timing Results

The performance of the program was tested against the two families of unary NFAs depicted in Figures 4.5 and 4.6. Tests were performed on UNIX SunOS 5.6 SPARC machines using the UNIX `time` command. The results are shown in Tables 4.1 and 4.2. Each timing test was executed 3 times, and the average time was recorded in the "System Time" column in the tables.

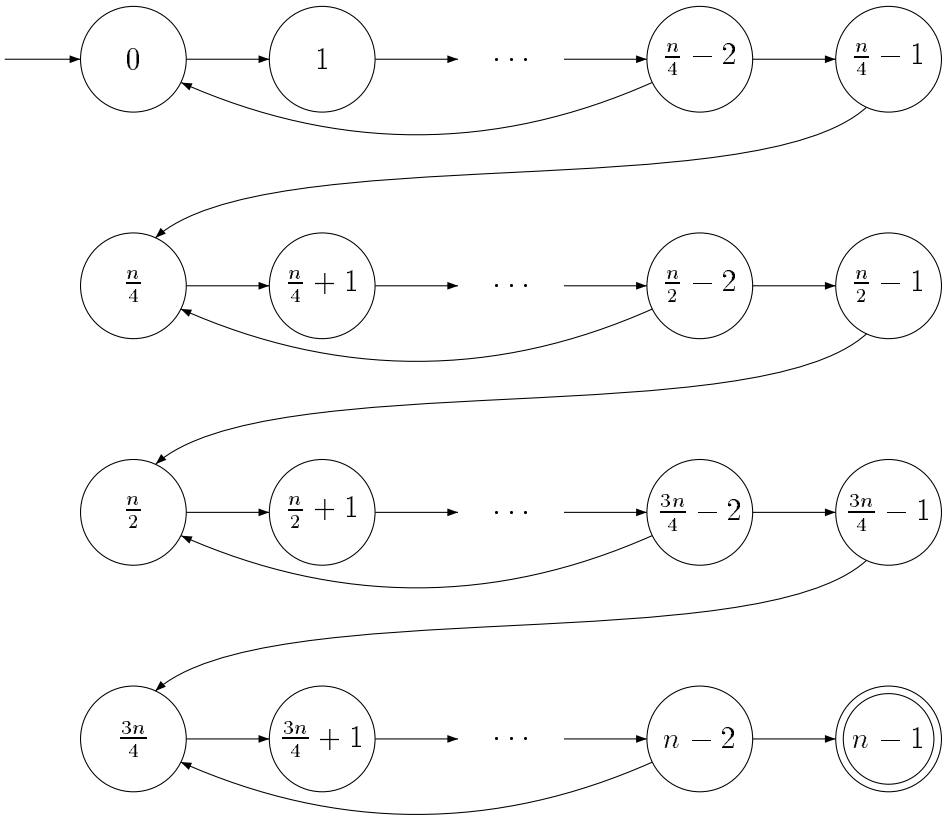

Figure 4.5: A family of unary NFAs.

The following function estimates the running time of the program on our computer (in

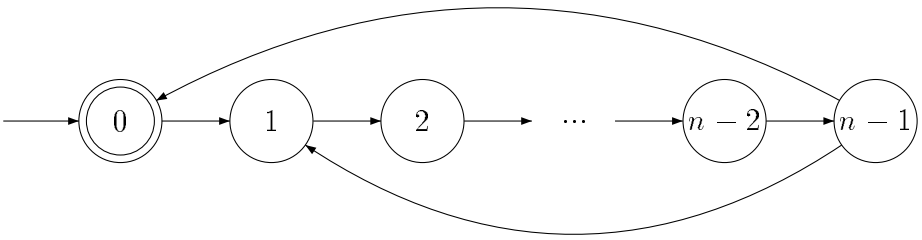

Figure 4.6: Another family of unary NFAs.

microseconds) on input $M = (Q, \{a\}, \delta, q_0, F)$:

$$F(M) = \frac{\sum_{i=1}^{k} (|T_i|^4) + m(n' + n + n'dk') + y(n'kk' + n + n'dk')}{38000}$$

where:

- $n = |Q|$;
- $m = n^2 + n$;
- $n' = \max_{0 \le i < m} |\delta(q_0, a^i)| \le n$;
- $d = \max_{q \in Q} |\delta(q, a)| \le n$;
- k is the number of SCCs in M;
- $k' = \max_{q \in Q} |S_q| \le k$;
- $|T_i| \le n$ is the number of states in T_i, the ith SCC of M; and
- $y = \max_{1 \le i \le k} y_i$, where $y_i \le |T_i|$ is the GCD of the lengths of the simple cycles in the SCC T_i.

The factor of 38000 is an estimate of the number of bit operations per microsecond performed by our computer. Observe that as n gets large, $F(M)$ is close to the actual running time of the program. Also observe that $F(M) \in O(kn^4)$, in accordance with Theorem 4.1.

Table 4.1: Expected and actual timing results for the unary NFAs in Figure 4.5, where $n' = \lceil n/9 \rceil, d = 2, k = 4, k' = 1, m = n^2 + n, |T_i| = n/4 - 1$ and $y = n/4 - 1$.

n	$F(M)$, where M has n states	System Time (μs)
20	0.35147	0.71
40	3.0809	1.78
60	11.8829	8.09
80	32.0313	24.66
100	71.1798	60.07
120	136.5074	126.47
140	238.5392	236.91
160	388.8646	414.85

Table 4.2: Expected and actual timing results for the unary NFAs in Figure 4.6, where $n' = n, d = 2, k = 1, k' = 1, m = n^2 + n, |T_i| = n$ and $y = 1$.

n	$F(M)$, where M has n states	System Time (μs)
10	0.3800	0.04
20	5.0968	1.28
30	24.2558	9.59
40	74.2779	36.44
50	177.9000	108.54
60	364.1747	267.93
70	668.4705	575.46
80	1132.4716	1121.61

4.7.3 Coding Difficulties and Bugs

The algorithms involving SCCs were the hardest to code. In particular, determining how to correctly take SCCs into account was quite tricky. It turns out that only keeping track

of SCCs which are passed through last without keeping track of i as well does not work, as difficulties arise if there exist two distinct directed paths leading to the same state but passing through different SCCs last. Most of the errors found during testing involved $S_{q,i}$.

Performance enhancements were also made along the way. For example, returning all final states in M' in one call instead of determining membership of a^i for each i separately (in Algorithm 4.3) reduced the overall running time of the program significantly.

Chapter 5

On Ambiguity in Unary NFAs

5.1 Introduction

Consider the problem of determining whether a given NFA is ambiguous or not. We will refer to this problem as the **ambiguity problem** for the rest of this chapter. Recall that an NFA M is said to be **ambiguous** if there exists a string $w \in L(M)$ such that M has two distinct accepting computations on w. An NFA is **unambiguous** if it is not ambiguous. An unambiguous finite automaton (UFA) is simply an unambiguous NFA.

We present the following additional definitions. Let $M = (Q, \Sigma, \delta, q_0, F)$ be an NFA and $w \in \Sigma^*$. We say that M is **ambiguous** on w if M has two distinct accepting computations on input w. We say that w is a **shortest ambiguous string** for M is M is ambiguous on w and M is not ambiguous on any string of length less than $|w|$. Note that shortest ambiguous strings are not necessarily unique.

Stearns and Hunt [27] gave an algorithm which decides whether a given NFA is ambiguous or not using $O(n^2)$ bit operations, where n is the number of states in the input. In this chapter, we will show two things. First, we will show that, in the worst case, shortest ambiguous strings for unary NFAs have length quadratic in the number of states of the NFA.

Second, we will consider the special case in which the input NFA is unary and in ChrNF (see Chapter 4), and give an algorithm to decide ambiguity which improves, in practice, on the algorithm presented by Stearns and Hunt. Once again, we fix $\Sigma = \{a\}$ for this entire chapter.

5.2 An Upper Bound on the Length of Shortest Ambiguous Strings Using Segments

In this section, we will show that shortest ambiguous strings for unary NFAs on n states have length at most $2n^2 + 1$. $M = (Q, \{a\}, \delta, q_0, F)$ is a unary NFA in the two following definitions.

Definition 5.1 *Let $w = a^k \in L(M)$. An **accepting path for** w **in** M is a sequence of states $[q_0, q_1, \ldots, q_k]$ where $q_i \in \delta(q_{i-1}, a)$ for each $1 \leq i \leq k$ and $q_k \in F$.*

Note that if M is unambiguous, M has exactly one accepting path for every $w \in L(M)$.

Definition 5.2 *Let $P = [q_0, q_1, \ldots, q_k]$ be an accepting path for some string in M, with each $q_i \in Q$. A **segment** of P is a subpath of the form $[q_i, q_{i+1}, \ldots, q_j]$, where $q_i = q_j$ and the states $\{q_i, q_{i+1}, q_{i+2}, \ldots, q_{j-1}\}$ are all distinct. Two segments $[q_{i_1}, q_{i_1+1}, \ldots, q_{i_2}]$, $[q_{j_1}, q_{j_1+1}, \ldots, q_{j_2}]$ of P, where $i_1 < j_1$ and $i_2 < j_2$, **overlap** if $i_2 > j_1$. The **length** of the segment $[q_i, q_{i+1}, \ldots, q_j]$ is $j - i$.*

Observe that segments of a given path P can overlap, but they cannot properly contain other segments. Also note that a path of length n contains $n + 1$ states. Lastly, we make three obvious, but useful, observations:

Observation 5.1 *(Lower Bound on Segment Length) Segments have length at least 1.*

Observation 5.2 *(Upper Bound on Segment Length) If an NFA M has n states and P is an accepting path in M, a segment of P has length at most n.*

Observation 5.3 *(Segment Removal) Let M be a unary NFA with n states, $a^k \in L(M)$, $P = [q_0, \ldots, q_{i-1}, q_i, q_{i+1}, \ldots, q_{j-1}, q_j, q_{j+1}, \ldots, q_k]$ be an accepting path for a^k in M, and $S = [q_i, q_{i+1}, \ldots, q_j]$ be a segment of P. Then the string a^{k-j+i} is accepted by M and has the accepting path $[q_0, q_1, \ldots, q_i, q_{j+1}, \ldots, q_k]$.*

We present a useful lemma before the main result of this section. The following elegant proof was given by Ming-Wei Wang.

Lemma 5.1 (The Submultiset Lemma) *Let S and T be multisets of elements from $\{1, 2, \ldots, n\}$, and suppose $|S| = |T| = n$. Then there exists a nonempty submultiset $S_1 \subseteq S$ and a nonempty submultiset $T_1 \subseteq T$ such that $\sum_{s \in S_1} s = \sum_{t \in T_1} t$.*

Proof. Let $S = \{s_1, s_2, \ldots, s_n\}$ and $T = \{t_1, t_2, \ldots, t_n\}$. Set $u_j = \sum_{i=1}^{j} s_i$ and $v_k = \sum_{i=1}^{k} t_i$; that is, the u_j and v_k are the running sums of S and T respectively. If $u_n = v_n$ then we are done; simply take $S_1 = S$ and $T_1 = T$. Otherwise, without loss of generality, we may assume that $u_n < v_n$. Now, for each u_j, there exists a v_k with $0 \leq v_k - u_j < n$, since each of the terms in the running sum differ by at most n. We obtain n such differences, one for each u_j. If one of these differences, say, $v_k - u_j$, equals 0, we are done — simply take $S_1 = \{s_1, s_2, \ldots, s_j\}$ and $T_1 = \{t_1, t_2, \ldots, t_k\}$. Otherwise, by the Pigeonhole Principle, two of these differences must be equal to each other. In this case, we have $v_k - u_j = v_{k'} - u_{j'}$, where $j \neq j'$. Without loss of generality, we may assume that $u_j > u_{j'}$, so that we have the equality $v_k - v_{k'} = u_j - u_{j'}$. This shows that the submultisets $S_1 = \{s_{j'+1}, s_{j'+2}, \ldots, s_j\}$ and $T_1 = \{t_{k'+1}, t_{k'+2}, \ldots, t_k\}$ have the same sum, which completes the proof. □

The Submultiset Lemma allows us to give a good upper bound on the length of shortest ambiguous strings. Suppose w is a shortest string accepted by M with at least two distinct

accepting computations and assume to the contrary that $|w|$ is large. We show that the two accepting paths for w must have at least a certain number of non-overlapping segments. This allows us to use the Submultiset Lemma to deduce that the sum of the lengths of some of the segments in the first path must equal the sum of the lengths of some of the segments in the second path. Thus, removing these segments yields two distinct accepting paths for a shorter string, which yields the desired contradiction. Here is the result:

Theorem 5.1 *If a unary NFA M on n states is ambiguous, then the shortest string accepted by M which has at least two different accepting computations has length at most $2n^2 + 1$.*

Proof. Let w be a shortest string accepted by M with at least two different accepting computations. Assume to the contrary that $|w| = 2n^2 + 2$. (It will be clear how to modify the argument to handle the case where $|w| > 2n^2 + 2$.) Then w has two distinct accepting paths P_1 and P_2 of length $2n^2 + 2$. Since these paths are distinct, they must differ in some state. This different state must appear either in the first $n^2 + 1$ states of P_1 and P_2 (not including the initial q_0), or the last $n^2 + 1$ states. Let Q_1 and Q_2 denote the first $n^2 + 1$ states in P_1 and P_2 respectively, and let R_1 and R_2 denote the last $n^2 + 1$ states in P_1 and P_2 respectively. Without loss of generality, we may assume that the different state appears in Q_1 and Q_2, so that Q_1 and Q_2 are distinct subpaths.

Since M has n states, the first $n + 1$ states of R_1 and R_2 must contain at least one non-overlapping segment. Similarly, the first $2n + 1$ states of R_1 and R_2 must contain at least two non-overlapping segments. Extending this argument to all of R_1 and R_2 implies that the $n^2 + 1$ states of R_1 and R_2 must contain at least n non-overlapping segments. Let $s_1, s_2, \ldots, s_n$ denote the lengths of the n non-overlapping segments in R_1, and let $t_1, t_2, \ldots, t_n$ denote the lengths of the similar segments in R_2.

By Observations 5.1 and 5.2, $S = \{s_1, s_2, \ldots, s_n\}$ and $T = \{t_1, t_2, \ldots, t_n\}$ are multisets of size n consisting of integers from 1 to n inclusive. Thus, by the Submultiset Lemma, there

exist submultisets $S_1 \subseteq S$ and $T_1 \subseteq T$ with the same sum. Remove the segments in R_1 corresponding to lengths in S_1 to obtain a new path R_3 and do the same for the segments in R_2 corresponding to lengths in T_1 to obtain a new path R_4. Consider the composite paths Q_1R_3 and Q_2R_4. These two composite paths are both accepting (by Observation 5.3), have the same length since the removed segments were assumed to be non-overlapping, and are strictly shorter than P_1 and P_2. Furthermore, these two paths are distinct, because it was assumed that Q_1 and Q_2 differed in at least one state. This implies that we have two distinct accepting paths in M for some string strictly shorter than w. This is a contradiction since w was assumed to be the shortest string with two distinct accepting paths in M. □

5.3 A Better Upper Bound on the Length of Shortest Ambiguous Strings

In 1985, Stearns and Hunt [27] gave an algorithm which decides whether a given NFA is ambiguous or not using $O(n^2)$ bit operations. Using the ideas presented in their proof of this result, we can prove the following upper bound on the length of shortest ambiguous strings for unary NFAs with n states:

Theorem 5.2 *If a unary NFA on n states is ambiguous, then the shortest string accepted by M which has at least two different accepting computations has length at most $\frac{n^2+3n}{2}$.*

Proof. Let $M = (Q, \{a\}, \delta, q_0, F)$ be the input NFA, where $|Q| = n$. We construct a second unary NFA, $M' = (Q', \{a\}, \delta', q_0', F')$, defined as follows:

- $Q' = (Q \times Q) \cup Q$;
- $q_0' = [q_0, q_0]$;

- $\delta'([q_1, q_2], a) = \{[u, v] | u \in \delta(q_1, a), v \in \delta(q_2, a)\}$
  $\cup \{q\}$, if $q \in \delta(q_1, a) \cap \delta(q_2, a)$ and $q_1 \neq q_2$;
- $\delta'(q, a) = \delta(q, a)$; and
- $F' = F$.

Observe that M' has two portions, a $Q \times Q$ portion and a Q portion. The $Q \times Q$ portion simulates the nondeterminism in M (but has no final states) while the Q portion is identical to M.

I claim that for any string $w = a^k \in \{a\}^*$, $w \in L(M')$ if and only if M is ambiguous on w. First, suppose that $w \in L(M')$, and consider an accepting path P for w in M'. Let q be the first state of this accepting path in the Q portion (such a state must exist by construction of M'), and suppose this state is the $(i+1)$st state in the accepting path. Then by construction of M', there exist at least two distinct paths of length i from q_0 to q in M. Since there is a path of length $k-i$ from q to some final state in M' (consider the last $k-i$ states of P), the same path exists in M and thus M is ambiguous on w. Conversely, suppose M is ambiguous on w. Let P_1 and P_2 be two distinct accepting paths for w in M. Since these two paths are distinct, they must diverge at some point and reconverge at a later point. Let q be the state in M where P_1 and P_2 first converge after diverging for the first time, and suppose this is the $(i+1)$st state in the two paths. Then $q \in \delta'(q_0', a^i)$ and hence $w \in L(M')$.

Now, let $x \in L(M')$ and let P be an accepting path for x in M'. By Observation 5.3, we can remove all of the segments from P to obtain a shorter accepting path P' for a string x', such that $|x'| < |(Q \times Q) \cup Q| = n^2 + n$. Then since $x' \in L(M')$, M is ambiguous on x', which shows that the shortest ambiguous string for M has length at most $n^2 + n$.

To obtain the upper bound of $\frac{n^2+3n}{2}$, observe that we do not need as many as $(Q \times Q) \cup Q$ states in the construction of M', since for all $q_1, q_2 \in Q$ with $q_1 \neq q_2$, the states $[q_1, q_2]$

and $[q_2, q_1]$ in Q' are indistinguishable. This allows us to remove $\frac{n^2-n}{2}$ states from Q' in our construction, leaving $\frac{n^2+3n}{2}$ states, from which the result follows. □

5.4 A Lower Bound on the Length of Shortest Ambiguous Strings

Note that the shortest string accepted by a unary NFA on n states with at least two different accepting computations could have length as great as $\frac{n^2-2n}{4}$; this is achieved by the unary NFA shown in Figure 5.1 (here n is even).

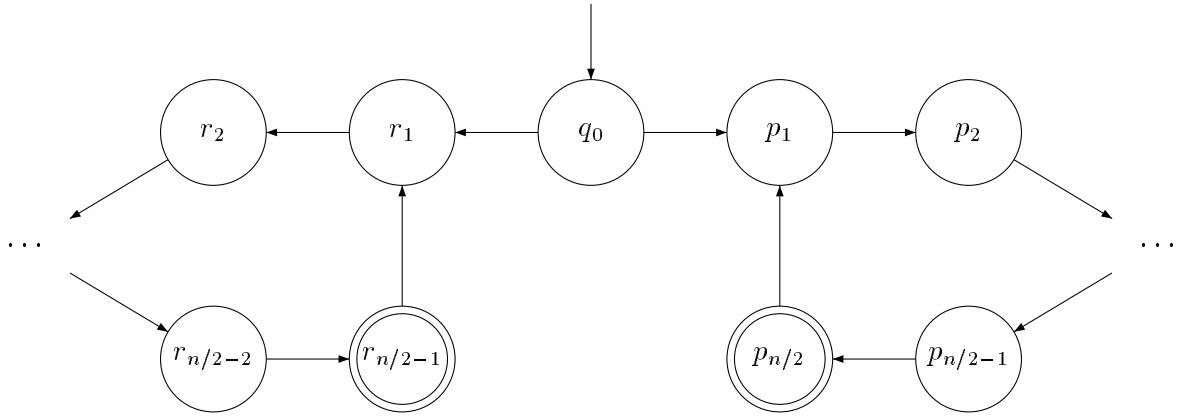

Figure 5.1: A unary NFA for which the shortest string with two different accepting computations has length $\frac{n^2-2n}{4}$. The loop with the p_i has length $n/2$ and the loop with the r_i has length $n/2 - 1$.

5.5 The Ambiguity Problem for Unary NFAs in Chrobak Normal Form

We now consider the ambiguity problem for unary NFAs in Chrobak normal form. Recall that a unary NFA is in ChrNF if it consists of an initial tail and some loops adjacent to the end of the tail (see Definition 4.1). It is clear that the contents of the loops, and not the contents of the tail, determine whether a unary NFA in ChrNF is ambiguous or not. Before we give an algorithm to decide the ambiguity problem for this special class of unary NFAs, we define the concept of loop condensation.

Definition 5.3 *Let $M = (Q, \{a\}, \delta, q_0, F)$ be a unary NFA in ChrNF and let $P = \{p_0, p_1, \ldots, p_{y-1}\}$ be a loop in M, where $\delta(p_i, a) = \{p_{i+1}\}$ for $0 \leq i \leq y-2$, $\delta(p_{y-1}, a) = \{p_0\}$, and p_0 is the state adjacent to the tail of M. Let $c \geq 1$ be an integer which divides y. We say that a loop R is a* ***condensation of P by a factor of*** *c if $R = \{r_0, r_1, \ldots, r_{y/c-1}\}$ is a loop, where $\delta(r_i, a) = \{r_{i+1}\}$ for $0 \leq i \leq y/c-2$, $\delta(r_{y/c-1}, a) = \{r_0\}$, r_0 is the state adjacent to the tail of M, and $r_i \in F$ if and only if $\exists j, 0 \leq j < y$ with $p_j \in F$ and $j \equiv i \bmod (y/c)$.*

Figure 5.2 shows an example of loop condensation. Loop R is a condensation of loop P by a factor of 2.

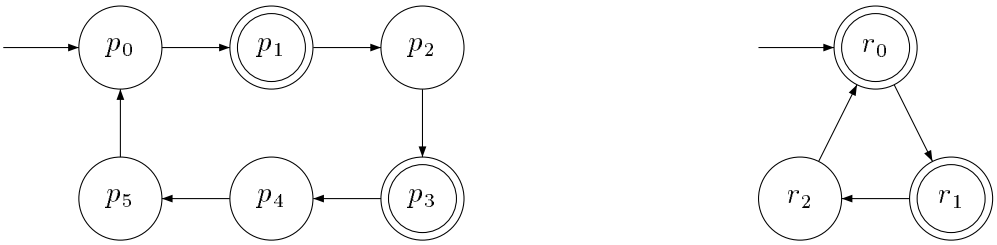

Figure 5.2: Loop P is on the left and loop R is on the right.

Here is the algorithm which decides the ambiguity problem for unary NFAs in ChrNF:

```
ALGORITHM 5.1
(Input: a unary NFA M = (Q, {a}, q_0, δ, F) in Chrobak normal form)
(Output: YES if M is ambiguous; NO otherwise)
begin
   k ← number of loops in M
   for i from 1 to k do
      P_i ← loop i
      y_i ← length of loop P_i
   end for
   for i from 1 to k − 1 do
      for j from i + 1 to k do
         d ← gcd(y_i, y_j)
         if d = 1 then
            if P_i and P_j both contain a final state then
               return YES
            end if
         else
            R_i ← P_i condensed by a factor of y_1/d (thus R_i has length d)
            R_j ← P_j condensed by a factor of y_2/d (similarly for R_j)
            if R_i and R_j contain a final state in the same position then
               return YES
            end if
         end if
      end for
```

```
   end for
   return NO
end
```

Theorem 5.3 *Algorithm 5.1 correctly decides whether a unary NFA M in Chrobak normal form is ambiguous using $O(k^2y)$ bit operations, where k is the number of loops in M and y is the number of states in the longest loop of M.*

Proof. M cannot be ambiguous if only one of its loops contains final states. Thus, having final states in two separate loops is a necessary (but not sufficient) condition for ambiguity. Let P_1 and P_2 be two loops in M which contain final states, and let y_1 and y_2 be their respective lengths. If $\gcd(y_1, y_2) = 1$, then M is ambiguous as there will be two distinct accepting paths (one ending in P_1, the other ending in P_2) for some string by the Chinese Remainder Theorem.

Now suppose that $d = \gcd(y_1, y_2) > 1$. It does not necessarily follow that M is ambiguous; however, we can use the technique of loop condensation to obtain two smaller loops R_1 and R_2 of length d. It remains to show that there are two distinct accepting paths in M ending in P_1 and P_2 if and only if R_1 and R_2 have a final state in the same position.

Suppose the former is true. Then the accepting paths must end in some final states p_{j_1} and p_{j_2} in P_1 and P_2, respectively, where p_{j_1} is j_1 states away from the initial state of P_1 and p_{j_2} is j_2 states away from the initial state of P_2. Then there must exist integers $m_1, m_2 \geq 0$ such that $m_1y_1 + j_1 = m_2y_2 + j_2$. Then $j_1 \equiv j_2 \bmod d$, since $d = \gcd(y_1, y_2)$. Let $0 \leq i < d$ be the unique integer satisfying $i \equiv j_1 \equiv j_2 \bmod d$. Then by definition of loop condensation, the ith state from the initial states in both R_1 and R_2 are final, as required.

Now, let us show the reverse implication. Suppose R_1 and R_2 have a final state in the same position, say, position i. Then by definition of loop condensation, there exist integers j_1 and j_2 such that the state in P_1 of distance j_1 from the initial state (call this

state p_{j_1}) is accepting, the state in P_2 of distance j_2 from the initial state (call this state p_{j_2}) is accepting, and $j_1 \equiv j_2 \equiv i \bmod d$. Without loss of generality, we may assume that $j_2 \geq j_1$. Thus $j_2 - j_1 = cd$ for some $c \in \mathbb{N}$. Now, since $\gcd(y_1, y_2) = d$, we can apply the Euclidean Algorithm to obtain integers $n_1, n_2 \in \mathbb{N}$ satisfying $n_1 y_1 - n_2 y_2 = d$. Multiplying this equation by c gives $cn_1 y_1 - cn_2 y_2 = cd = j_2 - j_1$. Let $m_1 = cn_1$ and $m_2 = cn_2$. Then $m_1 y_1 - m_2 y_2 = j_2 - j_1$, or $m_1 y_1 + j_1 = m_2 y_2 + j_2$. Since c, n_1 and n_2 are all nonnegative, m_1 and m_2 are nonnegative. This shows that there are two accepting paths of the same length in M, one ending in P_1 and one ending in P_2, as required. (For the first path, we travel around P_1 m_1 times before stopping at p_{j_1}; for the second path, we travel around P_2 m_2 times before stopping at p_{j_2}). This completes the proof of correctness of the algorithm.

As for the complexity of this algorithm, the factor of k^2 occurs when checking the pairwise GCD of the lengths of the loops in M. The factor of y arises when checking loops for the presence of final states. □

Lastly, since k and y in the previous theorem are both bounded above by n, the number of states in M, we have the following corollary:

Corollary 5.1 *There exists an algorithm to decide whether a unary NFA M in Chrobak normal form is ambiguous using $O(n^3)$ bit operations, where n is the number of states in M.*

Although the algorithm by Stearns and Hunt uses $O(n^2)$ bit operations, it happens that for most unary NFAs in ChrNF, k and y are much smaller than n and thus in practice, Algorithm 5.1 usually outperforms the algorithm by Stearns and Hunt.

5.6 Other Work in the Area

Study of ambiguity in NFAs is not restricted to determining whether a given NFA M is unambiguous or not. If for all $w \in L(M)$, there exist at most k accepting computations for

w, then M is said to be **ambiguous of degree** $\leq k$. M is **finitely ambiguous** if M is ambiguous of degree $\leq k$ for some $k \in \mathbb{N}$. M is **polynomially ambiguous** if there exists a polynomial $p(n)$ such that M is ambiguous of degree $\leq p(n)$, where n is the number of states in M. The term **exponentially ambiguous** is defined analogously. (Since there can be at most $|\Sigma|^n$ paths between two states in an NFA with n states, all NFAs are exponentially ambiguous.)

Mandel and Simon [20] introduced and studied the concept of ambiguity in NFAs, and showed that there exist algorithms to determine whether an NFA is finitely ambiguous, strictly polynomially ambiguous or strictly exponentially ambiguous.

Weber and Seidl [28] showed that the algorithms described by Mandel and Simon can be performed in time polynomial in the size of the input.

Stearns and Hunt [27] showed that the equivalence and containment problems for NFAs, known to be **PSPACE**-complete in general, can be solved efficiently when the NFAs are finitely ambiguous. (Given two NFAs M_1 and M_2, the equivalence problem for NFAs asks whether $L(M_1) = L(M_2)$; the containment problem asks whether $L(M_1) \subseteq L(M_2)$.) The authors also gave an algorithm to decide, for all $k \geq 1$, whether an NFA is ambiguous of degree $\leq k$ using $O(n^{k+1})$ bit operations, where n is the number of states in the NFA. Taking $k = 1$ gives an algorithm deciding the ambiguity problem using $O(n^2)$ bit operations. Furthermore, by slightly modifying Theorem 5.10 in their paper, an upper bound of $n^2 + n$ can be obtained on the length of the shortest ambiguous string accepted by an ambiguous NFA over an arbitrary alphabet. While this bound is better than the bound in Theorem 5.1 for unary alphabets, the proofs of the two results use completely different methods.

Ravikumar and Ibarra [23] and Leung [16] studied succinctness of various classes of ambiguous finite automata. Ravikumar and Ibarra showed that UFAs are exponentially more succinct than DFAs, meaning that for infinitely many n, there exists an n-state UFA for which the smallest equivalent DFA has 2^n states. They also showed that finitely ambigu-

ous NFAs were exponentially more succinct than UFAs. Leung showed that exponentially ambiguous NFAs are exponentially more succinct than polynomially ambiguous NFAs. It is still open whether polynomially ambiguous NFAs are exponentially more succinct than finitely ambiguous NFAs.

Chapter 6

State Complexity of the Minimization Operation

6.1 Introduction

We now examine problems dealing with state complexity of operations which preserve regularity. The concept of state complexity was introduced by Yu, Zhuang and Salomaa [30] and is a descriptional complexity measure for regular languages based on DFAs. Let L be a regular language. The **state complexity** of L, denoted $\mathrm{sc}(L)$, is the number of states in the minimal DFA accepting L. (We know, by the theorem of Myhill and Nerode [13, Theorem 3.9], that minimal DFAs are unique up to relabeling of states; this allows us to refer to *the* minimal DFA.)

We will be using the total and partial orders described in Definitions 3.1 and 3.2. For the rest of this chapter, we will fix $\Sigma = \{0, 1\}$, with $0 \prec 1$.

Let $L \subseteq \Sigma^*$ be a language (not necessarily regular). Then L_{min} is the subset of L consisting of the lexicographically smallest strings of each length in L. More formally, we

define:

$$L_{min} = \bigcup_{n \geq 0} \{x \in L \cap \Sigma^n : \forall y \in L \cap \Sigma^n, x \preceq y\}.$$

We refer to L_{min} as the **minimization** of L. For example, let L be the language of all palindromes over Σ^* beginning with the symbol 1. Then $L_{min} = 1 + 10^*1$.

Consider the following problem: given a language L, give upper and lower bounds on the state complexity on L_{min} in the worst case. It is known that the set of regular languages is closed under minimization; in his proof of this result, Shallit [25] gives an upper bound of $n2^{3n^2}$ on $\mathrm{sc}(L_{min})$. In this section, we will give a superpolynomial lower bound on $\mathrm{sc}(L_{min})$. This work in this section was done with Ming-Wei Wang.

6.2 A Superpolynomial Lower Bound on sc(L_{min})

Let p_i denote the ith prime ($p_1 = 2, p_4 = 7$, etc.), and consider the language

$$L = (10)^* + (110)^* + (11110)^* + \cdots + (1^{p_k - 1}0)^*,$$

where p_k is the greatest prime less than or equal to n. The finite automaton M, with its error state omitted, accepts L and is denoted in Figure 6.1.

We make the following observations:

Proposition 6.1 *The automaton M is deterministic, accepts L, and has asymptotically $\frac{n^2}{2 \log n}$ states.*

Proof. The first two claims are easily verified. Since the sum of the prime numbers less than or equal to n is asymptotically $\frac{n^2}{2 \log n}$ [2, p. 29] and p_k is the greatest prime less than or equal to n, the third claim follows. □

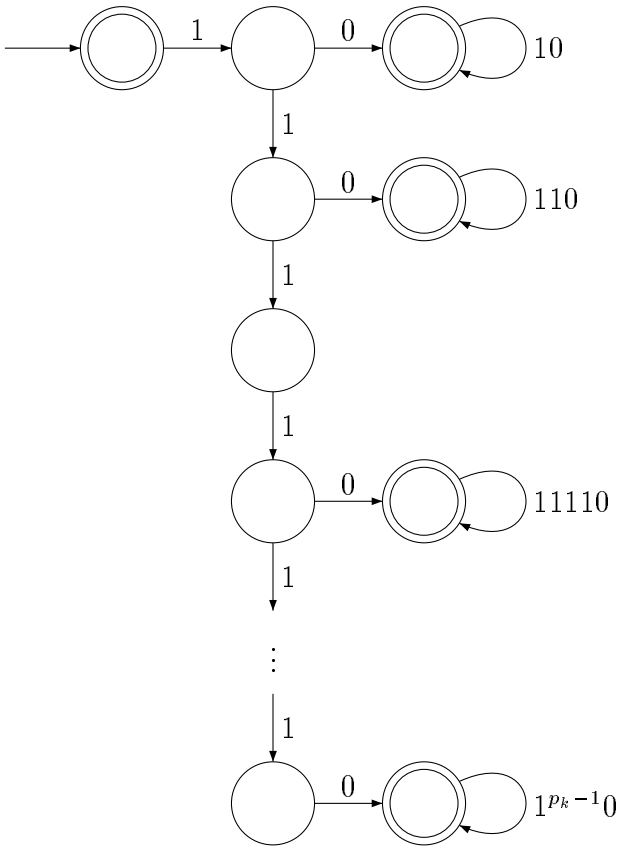

Figure 6.1: M, a DFA accepting L.

Proposition 6.2

$$\begin{aligned} L_{min} &= (10)^* + [(110)^{p_1}]^*(110) \\ &\quad +[(11110)^{p_1 p_2}]^*(11110 + (11110)^{p_1 p_2 - 1}) \\ &\quad + \cdots \\ &\quad +[(1^{p_k-1}0)^{p_1 p_2 \cdots p_{k-1}}]^*\Big(\sum_{\substack{1 \le j \le p_1 p_2 \cdots p_{k-1}, \\ \gcd(j, p_1 p_2 \cdots p_{k-1})=1}} (1^{p_k-1}0)^j\Big). \end{aligned}$$

Proof. Observe that any string $w \in L$ can be written in the form $w = x^m$, where x is of

the form $1^{p_i-1}0$ and $m \geq 0, 1 \leq i \leq k$. Thus, w belongs to L_{min} precisely when there are no other strings of length $|w|$ that can be written as a power of $1^{p_j-1}0$, where $j < i$. This occurs when the exponent m is coprime with $p_1 p_2 \cdots p_{i-1}$. □

Proposition 6.3 *Any DFA accepting L_{min} must have asymptotically at least $\prod_{i=1}^{k} p_i$ states.*

Proof. By Proposition 6.2, any DFA accepting L_{min} must accept $(1^{p_k-1}0)^m$ when $\gcd(m, \prod_{i=1}^{k-1} p_i) = 1$, but must reject $(1^{p_k-1}0)^m$ when $\gcd(m, \prod_{i=1}^{k-1} p_i) \neq 1$. Since there are $\prod_{i=1}^{k-1} p_i$ possible values for m (modulo $\prod_{i=1}^{k-1} p_i$) and the length of $1^{p_k-1}0$ is p_k, any DFA accepting L_{min} must have at least $\prod_{i=1}^{k} p_i$ states. □

Define $\vartheta(n)$ to be the logarithm of the product of the primes less than or equal to n; that is, $\vartheta(n) = \log(\prod_{p \leq n} p)$. This gives us the following result:

Theorem 6.1 *If $sc(L) = t$, then $sc(L_{min}) \sim e^{\sqrt{t \log t}(1+o(1))}$.*

Proof. If t denotes the number of states in M, then $t \sim \frac{n^2}{2 \log n}$ by Proposition 6.1. Then $t \log t \sim n^2$ and thus $\sqrt{t \log t} \sim n$. Since $\vartheta(n) = n(1 + o(1))$ [2, Corollary 8.2.7], we have that:

$$\begin{aligned} \mathrm{sc}(L_{min}) &\sim \prod_{i=1}^{k} p_i \text{ (by Proposition 6.3)} \\ &= \prod_{p \leq n} p \\ &= e^{\vartheta(n)} \\ &= e^{n(1+o(1))} \\ &\sim e^{\sqrt{t \log t}(1+o(1))}, \end{aligned}$$

as desired. □

6.3 Other Work in the Area

Yu, Zhuang and Salomaa [30] introduced worst-case state complexity as a measure of descriptional complexity. They determined the state complexities of the three basic operations on regular languages, union, concatenation and Kleene star, as well as intersection, quotient and reversal. Their analysis covers both the unary alphabet case and the general case.

Berstel and Boasson showed [3] that the set of context-free languages is also closed under the minimization operation described in this chapter. The proof of this result is much more involved than the corresponding proof for the regular language case, because the proof for the regular language case uses closure under complementation.

Birget [7] worked on a problem closely related to ours: given a partial order $<$, he computed upper and lower bounds on the state complexity of $\mathrm{MIN}(L)$, where $\mathrm{MIN}(L) = \{w \in L |$ there is no $x \in L$ such that $x < w\}$. However, none of the partial orders considered by Birget match our partial order $\prec$ and thus $\mathrm{MIN}(L)$ is not equal to L_{min} in general. Partial orders analyzed by Birget include the generalized lexical order, the generalized dictionary order and the generalized subsequence order. It is interesting that the results obtained by Birget differ substantially from ours, even though our partial orders differ only slightly.

Nicaud [21] investigated *average*-case state complexity of operations on regular languages, as opposed to *worst*-case state complexity. According to Nicaud, average-case analysis is extremely difficult since the exact number of non-isomorphic, connected DFAs with n states over a binary alphabet is not known. Thus, Nicaud focuses exclusively on languages and automata over unary alphabets.

Chapter 7

Conclusion and Open Problems

7.1 Conclusion

7.1.1 String Enumeration

String enumeration involves counting the exact number of strings which satisfy a given property. We showed that whether the problem of counting the exact number of strings of a given length in a language has an efficient solution or not depends on the model used to specify the language. The same holds true when considering the problem of computing the ith string in lexicographical order of a given length in a language.

7.1.2 Unary Nondeterministic Finite Automata

There exists a normal form for unary NFAs known as Chrobak normal form. We showed that there exists an efficient algorithm converting unary NFAs to ChrNF. Since regular expressions can be obtained easily from unary NFAs in ChrNF, we obtained an efficient method to convert an arbitrary unary NFA with n states to a regular expression of size $O(n^2)$. This method was coded in the C programming language.

7.1.3 State Complexity

The minimization operation, when applied to a language L, returns the subset of L consisting of the lexicographically smallest strings of each length in L. Since the regular languages are closed under minimization, it is natural to try and find good bounds on the state complexity of the minimization operation. We showed a superpolynomial lower bound on the state complexity of this operation.

7.2 Open Problems

We conclude by presenting a few open problems.

In Chapter 3, it is conjectured that the problem of computing the ith string in lexicographical order of length n in $L(G)$, for an unambiguous CFG G, can be solved efficiently via dynamic programming. Thus, we pose the following problem:

Open Problem 7.1 *Write down an algorithm which computes, given an unambiguous CFG G and integers $n \geq 1, i \geq 1$, the ith string in lexicographical order of length n in $L(G)$ efficiently.*

In Chapter 4, we show that for every unary NFA M on n states, there exists a regular expression r specifying $L(M)$ such that r has size $O(n^2)$. However, it is not known if there exist unary NFAs on n states which *require* a regular expression of size $\Omega(n^2)$. More precisely:

Open Problem 7.2 *Does there exist an n-state unary NFA M such that the smallest regular expression specifying $L(M)$ has size $\Omega(n^2)$?*

In section 5.2, an upper bound of $\frac{n^2+3n}{2}$ and a lower bound of $\frac{n^2-2n}{4}$ were given on the worst-case length of shortest ambiguous strings for unary NFAs. This leads to the following question:

Open Problem 7.3 *Can the worst-case upper and lower bounds on the length of shortest ambiguous strings as described in section 5.2 be improved?*

In section 6.2, a superpolynomial lower bound for $\mathrm{sc}(L_{min})$ was found, but no progress was made towards improving the best known upper bound of $n2^{3n^2}$. Thus, we pose the following open problem:

Open Problem 7.4 *Improve on the upper bound of $n2^{3n^2}$ for $sc(L_{min})$ as described in section 6.2.*

Appendix A

Source Code for Converting a Unary NFA to a Regular Expression

chrnf.c:

```
 1 #include <stdio.h>
 2 #include "chrnf.h"
 3 #include "list.h"
 4 #include "graph.h"
 5 #include "squarematrix.h"
 6 #include "unaryNFA.h"
 7 #include "regexp.h"
 8 #include "gcd.h"
 9
10 /*
11  * chrnf.c     V14.0
12  *
13  * Purpose:    Takes as input a unary NFA (from stdin) and
14  *             outputs (to stdout):
15  *             (1) a simplified unary NFA in Chrobak normal
16  *               form which accepts the same language; and
17  *             (2) a regular expression specifying the language
18  *               accepted by the input unary NFA.
19  *
20  * Author:     Andrew Martinez
```

```
21  *             Department of Computer Science
22  *             University of Waterloo
23  *             Waterloo, Ontario, Canada
24  *             N2L 3G1
25  *             aa2marti@math.uwaterloo.ca
26  *
27  * Reference:  M. Chrobak, Finite automata and unary languages,
28  *             TCS 47:149--158, 1986.
29  *
30  * Revision history:
31  *
32  * 27 Sep 2001 - in progress
33  * 10 Oct 2001 - completed the code which computes strongly
34  *               connected components, also completed the code
35  *               which takes a strongly connected graph as input
36  *               and returns the GCD of the lengths of all
37  *               cycles in the graph as output
38  * 12 Oct 2001 - added code to check if there exists an
39  *               accepting path P in M such that T_i is the last
40  *               strongly connected component traversed by P
41  * 17 Oct 2001 - changed IsAccepted to FindAcceptingStates
42  *               It now returns ALL final states in M_new
43  *               in one call, instead of returning a Y/N for a
44  *               particular input; this will reduce the running
45  *               time by O(n)
46  * 19 Oct 2001 - fixed bug in FindAcceptingStates in unaryNFA.c
47  * 31 Jan 2002 - removed debug comments
48  * 27 Mar 2002 - made some minor modifications, re-added debug
49  *               comments
50  * 27 Mar 2002 - V10.0 is the same as V9.5, except with all
51  *               debug comments removed
52  * 06 Apr 2002 - made a slight modification in
53  *               FindAcceptingStates - do not insert the sccID
54  *               of q_0 to the SCC lists if q_0 does not belong
55  *               to any SCC
56  * 06 Jun 2002 - program now simplifies the resulting unary NFA
57  *               in ChrNF by comparing final state status of the
58  *               last state of the tail and the last state of
59  *               the loops
60  * 07 Jun 2002 - program now outputs a regular expression r
61  *               specifying the language accepted by the input
62  *               unary NFA
```

```
 63  * 07 Jun 2002 - added regexp.c and regexp.h to modularize code
 64  * 10 Jun 2002 - added some comments;
 65  *               expressions of the form epsilon(r) are now
 66  *               simplified to (r)
 67  * 26 Jun 2002 - truncated lines over 64 columns in length for
 68  *               insertion into thesis appendix
 69  *
 70  * DISCLAIMER OF WARRANTY:  THE PRODUCT IS PROVIDED FREE OF
 71  *               CHARGE, AND, THEREFORE, ON AN "AS IS" BASIS,
 72  *               WITHOUT WARRANTY OF ANY KIND,EXPRESS OR
 73  *               IMPLIED, INCLUDING WITHOUT LIMITATION THE
 74  *               WARRANTIES THAT IT IS FREE OF DEFECTS, VIRUS
 75  *               FREE, ABLE TO OPERATE ON AN UNINTERRUPTED
 76  *               BASIS, MERCHANTABLE, FIT FOR A PARTICULAR
 77  *               PURPOSE OR NON-INFRINGING.
 78  */
 79
 80 int main(int argc, char *argv[]) {
 81
 82   /*
 83    * M = the input unary NFA
 84    * n = number of states in M
 85    * f = number of final states in M
 86    * b = represents a final state
 87    * m = n squared plus n
 88    * k = number of strongly connected components in M
 89    * t = number of transitions in M
 90    * a1, a2 = represent a transition
 91    * T = array of size k which is a list of strongly connected
 92    *       components in M, T[i] is a subgraph of the graph of M
 93    * y = array of size k, y[i] is the GCD of the lengths of all
 94    *       cycles in T[i]
 95    *
 96    * i,j = loop index
 97    * stateIndex = an index variable
 98    */
 99   int a1, a2, b, f, i, j, k, m, n, t, *y, stateIndex;
100   Graph **T;
101
102   unaryNFA *M;
103
104   /*
```

```
105    * 06 Jun 2002 - NEW
106    *   variables used for simplification algorithm
107    *   tailAcceptingStatus is an int array of size m;
108    *     tailAcceptingStatus[i] equals 1 if and only if q_i is a
109    *     final state
110    *   trueTailLength is the length of the tail after
111    *     simplification; this value could be anywhere between 1
112    *     and m, inclusive
113    *   loopAcceptingStatus is an array of k pointers; the ith
114    *     pointer points to an int array of size y_i;
115    *     loopAcceptingStatus[i][j] equals 1 if and only if p_i,j
116    *     is a final state, where 0 <= j < y_i
117    *   trueLoopHead is an int array of size k; trueLoopHead[i]
118    *     denotes the index of the true head of loop i after
119    *     simplification; this value could be anywhere between 0
120    *     and y_i-1, inclusive
121    */
122
123   int *tailAcceptingStatus;
124   int trueTailLength;
125   int **loopAcceptingStatus;
126   int *trueLoopHead;
127
128   /*
129    * Read in input
130    * Format of input file is critical - see README.txt and
131    * SAMPLEINPUT.txt for correct format
132    */
133
134   /* Read in n and initialize unaryNFA */
135   scanf("%d\n", &n);
136   unaryNFAInitialize(&M, n);
137
138   /* Read in t and add transitions to M */
139   scanf("%d\n", &t);
140
141   for (i=1; i<=t; i++) {
142
143     scanf("%d %d\n", &a1, &a2);
144     AddTransition(M, a1, a2);
145
146   } /*for*/
```

```
147
148   /* Read in f and add final states to M */
149   scanf("%d\n", &f);
150
151   for (i=1; i<=f; i++) {
152
153     scanf("%d\n", &b);
154     MakeFinalState(M, b);
155
156   } /*for*/
157
158   /*
159    * At this point, M has been read in completely
160    * Start the algorithm and output m
161    */
162
163   m = n*n+n;
164
165   /*
166    * Find the strongly connected components of M - there are
167    * potentially as many strongly connected components as there
168    * are states, so assume the worst
169    */
170
171   T = (Graph **)malloc(M->numStates*sizeof(int));
172   StronglyConnectedComponents(M, T, &k);
173
174   /*
175    * Now compute the GCD of lengths of cycles in the graphs in T
176    * and output each of the cycle lengths
177    */
178   y = (int *)malloc(k*sizeof(int));
179
180   for (i=0; i<k; i++) {
181
182     y[i] = FindGCDOfCycleLengths(T[i]);
183
184   } /*for*/
185
186   /* Allocate memory and initialize variables */
187   tailAcceptingStatus = (int *)malloc(m*sizeof(int));
188   trueTailLength = m;
```

```
189   loopAcceptingStatus = (int **)malloc(k*sizeof(int));
190   trueLoopHead = (int *)malloc(k*sizeof(int));
191
192   for (i=0; i<k; i++) {
193
194     loopAcceptingStatus[i] = (int *)malloc(y[i]*sizeof(int));
195     trueLoopHead[i] = 0;
196
197   } /*for*/
198
199   /*
200    * Determine final state status for all states in the new
201    * unary NFA -- we can call FindAcceptingStates to do this for
202    * us.  Information about final states is returned in
203    * tailAcceptingStatus and loopAcceptingStatus
204    */
205
206   FindAcceptingStates(M, m, k, y,
207                       tailAcceptingStatus,
208                       loopAcceptingStatus);
209
210   /*
211    * Now that we know what the final states are, simplify the
212    * unary NFA by calling ChrNFSimplification.
213    * The info about the simplified unary NFA is returned in
214    * trueTailLength and trueLoopHead
215    */
216
217   ChrNFSimplification(m, k, y,
218                       tailAcceptingStatus,
219                       &trueTailLength,
220                       loopAcceptingStatus,
221                       trueLoopHead);
222
223   /* Output the simplified unary NFA in ChrNF */
224   printf("m = %d\n", trueTailLength);
225   printf("k = %d\n", k);
226
227   for (i=0; i<k; i++) {
228
229     printf("y_%d = %d\n", i+1, y[i]);
230
```

```
231   } /*for*/
232
233   printf("The following states are final states:\n");
234
235   for (i=0; i<trueTailLength; i++) {
236
237     if (tailAcceptingStatus[i] == 1) {
238
239       printf("q_%d\n", i);
240
241     } /*if*/
242
243   } /*for*/
244
245   for (i=0; i<k; i++) {
246
247     for (j=0; j<y[i]; j++) {
248
249       if (loopAcceptingStatus[i]
250                               [(trueLoopHead[i]+j)%y[i]] == 1) {
251
252         printf("p_%d,%d\n", i+1, j);
253
254       } /*if*/
255
256     } /*for*/
257
258   } /*for*/
259
260   /*
261    * Output a regular expression for this simplified
262    *   unary NFA
263    */
264
265   ComputeRegExp(k, y, tailAcceptingStatus, trueTailLength,
266        loopAcceptingStatus, trueLoopHead);
267
268   /* Done - free allocated memory */
269   for (i=0; i<k; i++) {
270
271     GraphNuke(&(T[i]));
272     free(loopAcceptingStatus[i]);
```

```
273
274   } /*for*/
275
276   free(T);
277   free(y);
278   unaryNFANuke(&M);
279
280   free(trueLoopHead);
281   free(tailAcceptingStatus);
282   free(loopAcceptingStatus);
283
284   return 0;
285
286 } /*main*/
287
```

chrnf.h:

```
 1 #ifndef __CNF__
 2 #define __CNF__
 3
 4 /*
 5  * Set to 1 to print debug statements,
 6  * set to 0 to turn debug stmts off
 7  */
 8
 9 #define DEBUG 0
10
11 #endif
```

gcd.c:

```
 1 #include "gcd.h"
 2
 3
 4 /*********************************************************
 5  * gcd.c - Code to compute the greatest common divisor of two
 6  *         positive integers.
 7  *
 8  * 19 Sep 2001 - Completed
 9  *********************************************************/
10
```

```
11
12 /*
13  * PRIVATE - swaps m and n
14  */
15 int swap(int *m, int *n) {
16
17   int temp = *m;
18   *m = *n;
19   *n = temp;
20
21 } /*swap*/
22
23
24 /*
25  * Computes the greatest common divisor of two integers m and n
26  * using the Euclidean algorithm
27  */
28 int gcd(int m, int n) {
29
30   if (m < n) swap(&m, &n);
31
32   /* ASSERT:  m >= n */
33
34   if (m%n == 0) return n;
35
36   return gcd(n, m%n);
37
38 } /*gcd*/
39
```

gcd.h:

```
 1 #ifndef __GCD__
 2 #define __GCD__
 3
 4 /*
 5  * Computes the greatest common divisor of two integers m and n
 6  * Pre:  m, n >= 1
 7  */
 8 int gcd(int m, int n);
 9
10 #endif
```

graph.c:

```
 1 #include <stdlib.h>
 2 #include "chrnf.h"
 3 #include "gcd.h"
 4 #include "graph.h"
 5 #include "list.h"
 6 #include "squarematrix.h"
 7
 8
 9 /*************************************************************
10  * graph.c - Implementation of functions used to manipulate
11  *           the GRAPH data type.
12  *
13  * 24 Sep 2001 - in progress
14  * 10 Oct 2001 - completed DFS and FindInducedSubgraph,
15  * 10 Oct 2001 - completed FindGCDOfCycleLengths
16  * 12 Oct 2001 - added sccID so that vertices could be
17  *                 identified with their strongly connected
18  *                 component
19  *************************************************************/
20
21
22 /*
23  * Global variable used by DFS routine to compute discoveryTime
24  * and finishingTime
25  */
26 int time;
27
28
29 /*************************************************************
30  * Graph functions
31  *************************************************************/
32
33
34 /*
35  * PRIVATE - determines if vertex values are out of range
36  * Returns 1 if true, 0 if false
37  */
38 int VertexValuesOutOfRange(Graph *G, int u, int v) {
39
```

```
40 #if 0
41   printf("Graph::VertexValuesOutOfRange - Start\n");
42 #endif
43
44   if (u < 0 ||
45       v < 0 ||
46       u >= G->numVertices ||
47       v >= G->numVertices) return 1;
48
49   return 0;
50
51 } /*VertexValuesOutOfRange*/
52
53
54 /*
55  * PRIVATE - Finds the index i such that Array[i] = target and
56  *   returns i
57  * Returns -1 if target was not found in Array
58  */
59 int FindIndex(int *Array, int arraySize, int target) {
60
61   int i;
62
63   for (i=0; i<arraySize; i++) {
64
65     if (Array[i] == target) return i;
66
67   } /*for*/
68
69   return -1;
70
71 } /*FindIndex*/
72
73
74 /*
75  * PRIVATE - Finds the maximum value in Array and returns its
76  *             index
77  *           size is the number of elements in Array
78  * Pre:  size >= 1
79  */
80 int FindMaxIndex(int *Array, int size) {
81
```

```
 82   int i,
 83       index = 0,
 84       maxval = Array[0];
 85
 86   /* Find the index corresponding to the maximum value */
 87   for (i=1; i<size; i++) {
 88
 89     if (Array[i] > maxval) {
 90
 91       index = i;
 92       maxval = Array[i];
 93
 94     } /*if*/
 95
 96   } /*for*/
 97
 98   return index;
 99
100 } /*FindMaxIndex*/
101
102
103 /*
104  * PRIVATE - subroutine used by DFS
105  *   Uses algorithm in Cormen, Leiserson and Rivest, Sec. 23.5
106  *   *k denotes the strongly connected component ID
107  *
108  *   If flag is 1, returns a strongly connected component of G
109  *     in L as a list of vertices
110  */
111 int DFSVisit(Graph *G, int u, List *L, int flag, int *k) {
112
113   int v;
114
115   /* WHITE vertex u has just been discovered */
116   G->colour[u] = GRAY;
117
118   /* Update time */
119   time++;
120   G->discoveryTime[u] = time;
121
122   /* Explore all neighbours of u */
123   for (v=0; v<G->numVertices; v++) {
```

```
124
125     /* Explore edge (u,v) */
126     if (IsAdjacent(G, u, v)) {
127
128       if (G->colour[v] == WHITE) {
129
130         /* Predecessor of v is u - now recurse on v */
131         G->predecessor[v] = u;
132         DFSVisit(G, v, L, flag, k);
133
134       } /*if*/
135
136     } /*if*/
137
138   } /*for*/
139
140   /* BLACKen u; it is finished */
141   G->colour[u] = BLACK;
142   time++;
143   G->finishingTime[u] = time;
144
145   /* Add u to the list and set the sccID */
146   if (flag) {
147
148     ListInsert(L, u, 0);
149     G->sccID[u] = *k;
150
151   } /*if*/
152
153   return 0;
154
155 } /*DFSVisit*/
156
157
158 /*
159  * Initializes a graph
160  */
161 int GraphInitialize(Graph **G, int numVertices) {
162
163   int i;
164
165   *G = (Graph *)malloc(sizeof(Graph));
```

```
166   (*G)->numVertices   = numVertices;
167   (*G)->colour        = (int *)malloc(numVertices*sizeof(int));
168   (*G)->discoveryTime = (int *)malloc(numVertices*sizeof(int));
169   (*G)->finishingTime = (int *)malloc(numVertices*sizeof(int));
170   (*G)->predecessor   = (int *)malloc(numVertices*sizeof(int));
171   (*G)->sccID         = (int *)malloc(numVertices*sizeof(int));
172   ListInitialize(&(*G)->edges, DOUBLE);
173
174   /* Initialize the scc ID's to -1 */
175   for (i=0; i<numVertices; i++) {
176
177     (*G)->sccID[i] = -1;
178
179   } /*for*/
180
181   return 0;
182
183 } /*GraphInitialize*/
184
185
186 /*
187  * Deletes a graph
188  * Returns 0 if successful
189  */
190 int GraphNuke(Graph **G) {
191
192   ListNuke(&(*G)->edges);
193   free((*G)->colour);
194   free((*G)->discoveryTime);
195   free((*G)->finishingTime);
196   free((*G)->predecessor);
197   free((*G)->sccID);
198   free(*G);
199   return 0;
200
201 } /*GraphNuke*/
202
203
204 /*
205  * Returns 1 if (u,v) is an edge in G, 0 otherwise
206  */
207 int IsAdjacent(Graph *G, int u, int v) {
```

```
208
209 #if 0
210   printf("Graph::IsAdjacent - Start\n");
211 #endif
212
213   if (VertexValuesOutOfRange(G, u, v)) return -1;
214   return isElemInList(G->edges, u, v);
215
216 } /*IsAdjacent*/
217
218
219 /*
220  * Adds the edge (u,v) to G
221  */
222 int AddEdge(Graph *G, int u, int v) {
223
224   if (VertexValuesOutOfRange(G, u, v)) return -1;
225   return ListInsert(G->edges, u, v);
226
227 } /*AddEdge*/
228
229
230 /*
231  * Deletes the edge (u,v) from G
232  */
233 int DeleteEdge(Graph *G, int u, int v) {
234
235   if (VertexValuesOutOfRange(G, u, v)) return -1;
236   return ListDelete(G->edges, u, v);
237
238 } /*DeleteEdge*/
239
240
241 /*
242  * Computes the transpose of a graph:  given a graph G = (V,E),
243  *   the transpose G^T = (V,E^T) where
244  *   E^T = { (u,v) : (v,u) is in E) }
245  * The transpose is returned in OutputGraph
246  * Returns 0 if successful
247  */
248 int FindTranspose(Graph *InputGraph, Graph **OutputGraph) {
249
```

```
250   int i, j;
251   Graph *NewGraph;
252
253   GraphInitialize(&NewGraph, InputGraph->numVertices);
254
255   for (i=0; i<InputGraph->numVertices; i++) {
256
257     for (j=0; j<InputGraph->numVertices; j++) {
258
259       if (IsAdjacent(InputGraph, i, j)) {
260
261         AddEdge(NewGraph, j, i);
262
263       } /*if*/
264
265     } /*for*/
266
267   } /*for*/
268
269   *OutputGraph = NewGraph;
270   return 0;
271
272 } /*FindTranspose*/
273
274
275 /*
276  * Performs a depth-first search on G;
277  *   computes the discovery time and finishing time for each
278  *   vertex in G
279  * Vertices are always considered in order of decreasing
280  *   finishing time
281  * Uses algorithm in Cormen, Leiserson and Rivest, Section 23.5
282  *
283  * If flag is 1, returns the strongly connected components of G
284  *   in GraphArray; return value of k is then the number of
285  *   strongly connected components in G
286  * In this event, must allocate memory for GraphArray before
287  *   calling
288  */
289 int DFS(Graph *G, Graph **GraphArray, int flag, int *k) {
290
291   int i, u;
```

```
292   int *tempArray;
293   List *L;
294
295   /* Reset *k to 0 */
296   *k = 0;
297
298   /* Initialize all vertices to WHITE and predecessors to -1 */
299   for (u=0; u<G->numVertices; u++) {
300
301     G->colour[u] = WHITE;
302     G->predecessor[u] = -1;
303
304   } /*for*/
305
306   /* Initialize global variable */
307   time = 0;
308
309   /* Set up a secondary array */
310   tempArray = (int *)malloc(G->numVertices*sizeof(int));
311
312   for (i=0; i<G->numVertices; i++) {
313
314     tempArray[i] = G->finishingTime[i];
315
316   } /*for*/
317
318   for (i=0; i<G->numVertices; i++) {
319
320     /*
321      * Consider the vertices in DECREASING finishing time
322      * Make sure that after finding the max index, set it to -1
323      * so that it is no longer in consideration - finishing
324      * times are always non-negative
325      */
326
327     u = FindMaxIndex(tempArray, G->numVertices);
328     tempArray[u] = -1;
329
330     /*
331      * At this point, u is the vertex with the largest finishing
332      * time
333      */
```

```
334
335     if (G->colour[u] == WHITE) {
336
337       /* Initialize a fresh list and visit vertex u */
338       ListInitialize(&L, SINGLE);
339       DFSVisit(G, u, L, flag, k);
340
341       /*
342        * At this point there are two possibilities:
343        * (1) L consists of one or more vertices forming a
344        *     strongly connected component in G
345        * (2) L consists of a single vertex which is not in any
346        *     strongly connected component in G
347        * Proceed if we are in case (1) and flag == 1
348        */
349       if (flag && (numElemsInList(L) != 1 ||
350                    IsAdjacent(G, u, u))) {
351
352         /*
353          * Get the induced subgraph of G based on the vertices
354          *   in L and store it in an entry of GraphArray
355          * Also increment the # of strongly connected components
356          *   in G
357          */
358
359         GraphInitialize(&(GraphArray[*k]), numElemsInList(L));
360         FindInducedSubgraph(G, L, &(GraphArray[*k]));
361         *k += 1;
362
363       } /*if*/
364       else {
365
366         /*
367          * Second case - implies vertex u is not in any strongly
368          * connected component - thus, reset its sccID
369          */
370         G->sccID[u] = -1;
371
372       } /*else*/
373
374       /* Delete list */
375       ListNuke(&L);
```

```
376
377     } /*if*/
378
379   } /*for*/
380
381   /* Free memory */
382   free(tempArray);
383   return 0;
384
385 } /*DFS*/
386
387
388 /*
389  * Returns the adjacency matrix for the graph G
390  * Adjacency matrix is returned in M
391  * Returns 0 if successful
392  */
393 int FindAdjacencyMatrix(Graph *G, SquareMatrix **M) {
394
395   int u, v;
396   SquareMatrix *NewMatrix;
397
398   SquareMatrixInitialize(&NewMatrix, G->numVertices);
399
400   for (u=0; u<G->numVertices; u++) {
401
402     for (v=0; v<G->numVertices; v++) {
403
404       if (IsAdjacent(G, u, v)) {
405
406         UpdateEntryInSquareMatrix(NewMatrix, u, v, 1);
407
408       } /*if*/
409
410     } /*for*/
411
412   } /*for*/
413
414   *M = NewMatrix;
415   return 0;
416
417 } /*FindAdjacencyMatrix*/
```

```
418
419
420 /*
421  * Returns the subgraph of InputGraph induced by the vertices
422  * in VertexList
423  * Subgraph is returned in OutputGraph
424  * Returns 0 if successful
425  *
426  * NOTE:  Since vertices in graphs are always numbered from 0 to
427  * n-1, the vertices in OutputGraph will be renumbered unless
428  * all n vertices of InputGraph are in VertexList.  In
429  * particular, vertex i in OutputGraph may not be the same
430  * vertex as vertex i in InputGraph.
431  */
432 int FindInducedSubgraph(Graph *InputGraph,
433                         List *VertexList,
434                         Graph **OutputGraph) {
435
436   int i, j, numElems, *VertexArray;
437   Graph *NewGraph;
438
439   /* Create a new graph */
440   GraphInitialize(&NewGraph, numElemsInList(VertexList));
441
442   /* Make use of VertexArray to store vertices */
443   numElems = numElemsInList(VertexList);
444   VertexArray = (int *)malloc(sizeof(int)*numElems);
445   j = 0;
446
447   for (i=0; i<InputGraph->numVertices; i++) {
448
449     if (isElemInList(VertexList, i, 0)) {
450
451       VertexArray[j] = i;
452       j++;
453
454     } /*if*/
455
456   } /*for*/
457
458   /* Add the edges */
459   for (i=0; i<InputGraph->numVertices; i++) {
```

```
460
461     for (j=0; j<InputGraph->numVertices; j++) {
462
463       if (isElemInList(VertexList, i, 0) &&
464           isElemInList(VertexList, j, 0) &&
465           IsAdjacent(InputGraph, i, j)) {
466
467         AddEdge(NewGraph,
468                 FindIndex(VertexArray, numElems, i),
469                 FindIndex(VertexArray, numElems, j));
470
471       } /*if*/
472
473     } /*for*/
474
475   } /*for*/
476
477   /* Free memory */
478   free(VertexArray);
479   *OutputGraph = NewGraph;
480   return 0;
481
482 } /*FindInducedSubgraph*/
483
484
485 /*
486  * Takes as input a strongly connected directed graph G and
487  * returns the GCD of the lengths of all cycles in G
488  */
489 int FindGCDOfCycleLengths(Graph *G) {
490
491   List *S;              /* S stores the cycle lengths     */
492   SquareMatrix *A;      /* A is the adjacency matrix of G */
493   SquareMatrix *B;      /* temporary matrix */
494   int temp, temp2;      /* temporary values */
495   int i, j;             /* loop indices     */
496   int g;                /* the GCD          */
497
498   ListInitialize(&S, SINGLE);
499   FindAdjacencyMatrix(G, &A);
500
501   /*
```

```
502    * FACT:  If A^i has a non-zero entry on its diagonal,
503    *        then G has a cycle of length i
504    */
505   for (i=1; i<=G->numVertices; i++) {
506
507     Exponentiate(A, i, &B);
508
509     /* Check the diagonal of B */
510     for (j=0; j<B->dimension; j++) {
511
512       GetEntryInSquareMatrix(B, j, j, &temp);
513       if (temp != 0) {
514
515         /* G has a cycle of length i - add i to S */
516         ListInsert(S, i, 0);
517         break;
518
519       } /*if*/
520
521     } /*for*/
522
523   } /*for*/
524
525   /*
526    * Compute the GCD of elements in S pairwise
527    * We know for sure that S is nonempty because G is strongly
528    *   connected
529    */
530
531   ReturnLastElemInList(S, &g, &temp);
532
533   while (numElemsInList(S) != 0) {
534
535     ReturnLastElemInList(S, &temp, &temp2);
536     g = gcd(g, temp);
537
538     if (g == 1) break;
539
540   } /*while*/
541
542   ListNuke(&S);
543   SquareMatrixNuke(&A);
```

```
  SquareMatrixNuke(&B);
  return g;

} /*FindGCDOfCycleLengths*/

```

graph.h:

```
#ifndef __GRAPH__
#define __GRAPH__

#include "squarematrix.h"
#include "list.h"

#define WHITE 0
#define GRAY 1
#define BLACK 2



/*************************************************************
 * GRAPH data structure
 * If graph has n vertices, vertices are numbered 0, 1, ... n-1
 *************************************************************/

typedef struct Graph_t {
  int numVertices;
  int *colour;         /* used in DFS; either WHITE, GRAY or
                          BLACK -- it's an array of size
                          numVertices */
  int *discoveryTime;  /* array of size numVertices */
  int *finishingTime;  /* array of size numVertices */
  int *predecessor;    /* array of size numVertices */
  int *sccID;          /* which strongly connected component a
                          particular vertex belongs to */
  List *edges;
} Graph;

/*************************************************************
 * Graph operations
 *************************************************************/
```

```
35
36 /*
37  * Initializes a graph
38  * G starts with no edges
39  */
40 int GraphInitialize(Graph **G, int numVertices);
41
42 /*
43  * Deletes a graph
44  * Returns 0 if successful
45  */
46 int GraphNuke(Graph **G);
47
48 /*
49  * Returns 1 if (u,v) is an edge in G, 0 otherwise
50  * Returns -1 if vertex values are not in the range
51  *   [0..numVertices-1]
52  */
53 int IsAdjacent(Graph *G, int u, int v);
54
55 /*
56  * Adds the edge (u,v) to G
57  * Returns -1 if vertex values are not in the range
58  *   [0..numVertices-1]
59  */
60 int AddEdge(Graph *G, int u, int v);
61
62 /*
63  * Deletes the edge (u,v) from G
64  * Returns -1 if vertex values are not in the range
65  *   [0..numVertices-1]
66  */
67 int DeleteEdge(Graph *G, int u, int v);
68
69 /*
70  * Computes the transpose of a graph:  given a graph G = (V,E),
71  * the transpose G^T = (V,E^T) where E^T = { (u,v) : (v,u) is
72  *   in E) }
73  * The transpose is returned in OutputGraph
74  * Returns 0 if successful
75  */
76 int FindTranspose(Graph *InputGraph, Graph **OutputGraph);
```

```
 77
 78 /*
 79  * Performs a depth-first search on G;
 80  * computes the discovery time and finishing time for each
 81  *   vertex in G
 82  * Vertices are always considered in order of decreasing
 83  *   finishing time
 84  * Uses algorithm in Cormen, Leiserson and Rivest, Section 23.5
 85  *
 86  * If flag is 1, returns the strongly connected components of G
 87  *   in GraphArray; return value of k is then the number of
 88  *   strongly connected components in G
 89  * In this event, must allocate memory for GraphArray before
 90  *   calling
 91  */
 92 int DFS(Graph *G, Graph **GraphArray, int flag, int *k);
 93
 94 /*
 95  * Returns the adjacency matrix for the graph G
 96  * Adjacency matrix is returned in M
 97  * Returns 0 if successful
 98  */
 99 int FindAdjacencyMatrix(Graph *G, SquareMatrix **M);
100
101 /*
102  * Returns the subgraph of InputGraph induced by the vertices
103  * in VertexList
104  * Subgraph is returned in OutputGraph
105  * Returns 0 if successful
106  *
107  * NOTE:  Since vertices in graphs are always numbered from 0 to
108  *   n-1, the vertices in OutputGraph will be renumbered unless
109  *   all n vertices of InputGraph are in VertexList.  In
110  *   particular, vertex i in OutputGraph may not be the same
111  *   vertex as vertex i in InputGraph.
112  */
113 int FindInducedSubgraph(Graph *InputGraph,
114                         List *VertexList,
115                         Graph **OutputGraph);
116
117
118 /*
```

```
 119  * Takes as input a strongly connected directed graph G and
 120  * returns the GCD of the lengths of all cycles in G
 121  */
 122 int FindGCDOfCycleLengths(Graph *G);
 123
 124
 125 #endif
 126
```

list.c:

```
   1 #include <stdlib.h>
   2 #include "list.h"
   3
   4
   5 /*************************************************************
   6  * list.c - Implementation of functions used to manipulate
   7  *          the LIST data type.
   8  *
   9  * 20 Sep 2001 - completed List functions
  10  * 10 Oct 2001 - completed ReturnLastElemInList
  11  *************************************************************/
  12
  13
  14 /*************************************************************
  15  * List functions
  16  *************************************************************/
  17
  18
  19 /*
  20  * Initializes an List
  21  */
  22 int ListInitialize(List **L, int listType) {
  23
  24   /* Add dummy node at head */
  25   *L = (List *)malloc(sizeof(List));
  26   (*L)->type = listType;
  27   (*L)->firstItem = -1;
  28   (*L)->secondItem = -1;
  29   (*L)->next = NULL;
  30   return 0;
  31
```

```
32 } /*ListInitialize*/
33
34
35 /*
36  * Deletes a list
37  */
38 int ListNuke(List **L) {
39
40   List *temp = *L;
41
42   while (*L != NULL) {
43
44     *L = (*L)->next;
45     free(temp);
46     temp = *L;
47
48   } /*while*/
49
50   return 0;
51
52 } /*ListNuke*/
53
54
55 /*
56  * Returns 1 if L is empty, 0 otherwise
57  */
58 int isListEmpty(List *L) {
59
60   /*
61    * L is empty if it consists of a single dummy node at the
62    * head
63    */
64
65   if (L->next == NULL) return 1;
66
67   return 0;
68
69 } /*isListEmpty*/
70
71
72 /*
73  * If L is a single element list, returns 1 if firstItem is in
```

```
 74  *   L, 0 otherwise
 75  * If L is a single element list, secondItem is ignored
 76  * If L is a double element list, returns 1 if
 77  *   (firstItem, secondItem) is in L, 0 otherwise
 78  * If L is a single element list, secondItem is ignored
 79  */
 80 int isElemInList(List *L, int first, int second) {
 81
 82   List *temp = L;
 83
 84   /* Reject if first < 0 or second < 0 */
 85   if (first < 0 || second < 0) return 0;
 86
 87   if (isListEmpty(L)) return 0;
 88
 89   while (temp != NULL) {
 90
 91     if (L->type == SINGLE && temp->firstItem == first) return 1;
 92     if (L->type == DOUBLE && temp->firstItem == first
 93         && temp->secondItem == second)
 94       return 1;
 95
 96     temp = temp->next;
 97
 98   } /*while*/
 99
100   return 0;
101
102 } /*isElemInList*/
103
104
105 /*
106  * Returns the number of single elements in a single list, or
107  * the number of paired elements in a paired list
108  */
109 int numElemsInList(List *L) {
110
111   int numElems = 0;
112   List *temp = L->next;
113
114   while (temp != NULL) {
115
```

```
116     numElems++;
117     temp = temp->next;
118
119   } /*while*/
120
121   return numElems;
122
123 } /*numElemsInList*/
124
125
126 /*
127  * Inserts element firstItem into L if L is a single element
128  *   list, or inserts (firstItem, secondItem) into L if L is a
129  *   double element list
130  * Returns 0 if insertion was successful
131  * Returns 1 if (firstItem, secondItem) is already in L
132  * Returns -1 if either firstItem or secondItem < 0
133  */
134 int ListInsert(List *L, int first, int second) {
135
136   List *temp = L, *prev;
137
138   /* Reject if first < 0 or second < 0 */
139   if (first < 0 || second < 0) return -1;
140
141   /* Check if elem is already in L */
142   if (isElemInList(L, first, second)) return 1;
143
144   /* Insert elem at end of list */
145   prev = temp;
146   temp = temp->next;
147
148   while (temp != NULL) {
149
150     prev = temp;
151     temp = temp->next;
152
153   } /*while*/
154
155   temp = (List *)malloc(sizeof(List));
156   temp->firstItem = first;
157   temp->secondItem = second;
```

```
158   temp->next = NULL;
159   prev->next = temp;
160   return 0;
161
162 } /*ListInsert*/
163
164
165 /*
166  * Deletes element elem from L
167  * Returns 0 if deletion was successful
168  * Returns 1 if elem was not found in L
169  * Returns -1 if elem < 0 or deletion was unsuccessful
170  * If L is a single element list, secondItem is ignored
171  */
172 int ListDelete(List *L, int first, int second) {
173
174   List *temp = L, *prev;
175
176   /* Reject if first < 0 or second < 0 */
177   if (first < 0 || second < 0) return -1;
178
179   /* Make sure elem is in L before proceeding */
180   if (!isElemInList(L, first, second)) return 1;
181
182   /* General case */
183   prev = temp;
184   temp = temp->next;
185
186   while (temp != NULL) {
187
188     if (L->type == SINGLE && temp->firstItem == first) {
189
190       prev->next = temp->next;
191       free(temp);
192       return 0;
193
194     } /*if*/
195     else if (L->type == DOUBLE && temp->firstItem == first
196         && temp->secondItem == second) {
197
198       prev->next = temp->next;
199       free(temp);
```

```
200       return 0;
201
202     } /*else if*/
203     else {
204
205       prev = temp;
206       temp = temp->next;
207
208     } /*else*/
209
210   } /*while*/
211
212   /* Something is wrong */
213   return -1;
214
215 } /*ListDelete*/
216
217
218 /*
219  * Returns the last element in L in firstItem and secondItem,
220  *   and also deletes this element from L
221  * Returns 0 if successful
222  * Returns -1 if L has no elements
223  * If L is a single element list, secondItem is ignored
224  */
225 int ReturnLastElemInList(List *L,
226                          int *firstItem,
227                          int *secondItem) {
228
229   List *temp = L;
230
231   /* Make sure list has non-zero number of elements */
232   if (numElemsInList(L) == 0) return -1;
233
234   /* Go to end of list */
235   while (temp->next != NULL) {
236
237     temp = temp->next;
238
239   } /*while*/
240
241   *firstItem = temp->firstItem;
```

```
  242   *secondItem = temp->secondItem;
  243   ListDelete(L, temp->firstItem, temp->secondItem);
  244   return 0;
  245
  246 } /*ReturnLastElemInList*/
  247
```

list.h:

```
    1 #ifndef __LIST__
    2 #define __LIST__
    3
    4 #define SINGLE 1  /* Single element lists */
    5 #define DOUBLE 2  /* Double element lists */
    6
    7
    8
    9
   10 /*************************************************************
   11  * LIST data structure
   12  * Note that a list can store up to two elements
   13  * Note also that this list may not store negative integers
   14  *************************************************************/
   15
   16 typedef struct List_t {
   17   int type;        /* SINGLE for single-item lists,
   18                       DOUBLE for double-item lists */
   19   int firstItem;   /* firstItem >= 0 */
   20   int secondItem;  /* secondItem >= 0, only relevant
   21                       if type == DOUBLE */
   22   struct List_t *next;
   23 } List;
   24
   25 /*************************************************************
   26  * List operations
   27  *************************************************************/
   28
   29 /*
   30  * Initializes a List
   31  */
   32 int ListInitialize(List **L, int listType);
   33
```

```
34 /*
35  * Deletes a list, frees memory before termination
36  */
37 int ListNuke(List **L);
38
39 /*
40  * Returns 1 if L is empty, 0 otherwise
41  */
42 int isListEmpty(List *L);
43
44 /*
45  * If L is a single element list, returns 1 if firstItem is in
46  *   L, 0 otherwise
47  * If L is a single element list, secondItem is ignored
48  * If L is a double element list, returns 1 if
49  *   (firstItem, secondItem) is in L, 0 otherwise
50  * If L is a single element list, secondItem is ignored
51  */
52 int isElemInList(List *L, int firstItem, int secondItem);
53
54 /*
55  * Returns the number of single elements in a single list, or
56  * the number of paired elements in a paired list
57  */
58 int numElemsInList(List *L);
59
60 /*
61  * Inserts element firstItem into L if L is a single element
62  *   list, or inserts (firstItem, secondItem) into L if L is a
63  *   double element list
64  * Returns 0 if insertion was successful
65  * Returns 1 if (firstItem, secondItem) is already in L
66  * Returns -1 if either firstItem or secondItem < 0
67  * If L is a single element list, secondItem is ignored
68  */
69 int ListInsert(List *L, int firstItem, int secondItem);
70
71 /*
72  * Deletes element firstItem from L if L is a single element
73  *   list, or deletes (firstItem, secondItem) from L if L is a
74  *   double element list
75  * Returns 0 if deletion was successful
```

```
 76  * Returns 1 if elem was not found in L
 77  * Returns -1 if either elem < 0 or deletion was unsuccessful
 78  * If L is a single element list, secondItem is ignored
 79  */
 80 int ListDelete(List *L, int firstItem, int secondItem);
 81
 82 /*
 83  * Returns the last element in L in firstItem and secondItem,
 84  *   and also deletes this element from L
 85  * Returns 0 if successful
 86  * Returns -1 if L has no elements
 87  * If L is a single element list, secondItem is ignored
 88  */
 89 int ReturnLastElemInList(List *L,
 90                          int *firstItem,
 91                          int *secondItem);
 92
 93
 94 #endif
 95
```

regexp.c:

```
  1 #include <stdlib.h>
  2 #include "chrnf.h"
  3
  4
  5 /*********************************************************
  6  * regexp.c    - Various functions used to output a regular
  7  *               expression specifying a given unary NFA.
  8  *
  9  * 06 Jun 2002 - program now simplifies the resulting unary NFA
 10  *               in ChrNF by comparing final state status of the
 11  *               last state of the tail and the last state of
 12  *               the loops
 13  * 07 Jun 2002 - program now outputs a regular expression r
 14  *               specifying the language accepted by the input
 15  *               unary NFA
 16  * 10 Jun 2002 - expressions of the form epsilon(r) are now
 17  *               simplified to (r)
 18  *
 19  *********************************************************/
```

```
20
21
22 /*************************************************************
23  * regexp functions
24  *************************************************************/
25
26
27 /*
28  * NEW - 06 Jun 2002
29  *
30  * This procedure accepts a unary NFA in ChrNF and simplifies it
31  * by eliminating as many states in the tail as possible.
32  *
33  * Input:
34  *   tailAcceptingStatus is an int array of size m;
35  *     tailAcceptingStatus[i] equals 1 if and only if q_i is a
36  *     final state
37  *   loopAcceptingStatus is an array of k pointers; the ith
38  *     pointer points to an array of size y_i;
39  *     loopAcceptingStatus[i][j] equals 1 if and only if p_i,j
40  *     is a final state, where 0 <= j < y_i
41  *   trueLoopHead is an int array of size k
42  *
43  * Output:
44  *   trueTailLength is the length of the tail after
45  *     simplification; this value could be anywhere between
46  *     1 and m, inclusive
47  *   trueLoopHead[i] denotes the index of the true head of
48  *     loop i after simplification; this value could be anywhere
49  *     between 0 and y_i-1, inclusive
50  */
51 int ChrNFSimplification(int m, int k, int *y,
52                         int *tailAcceptingStatus,
53                         int *trueTailLength,
54                         int **loopAcceptingStatus,
55                         int *trueLoopHead) {
56
57   int i, j;
58   int simplify;
59
60   /*
61    * Repeat until either the tail has length 1 or no further
```

```
62    * simplification is possible
63    */
64
65   for (i=m-1; i>=1 ;i--) {
66
67     simplify = 0;
68
69     /*
70      * If the last state in the tail is final and at least one
71      * of the last states in the k loops is also final, then
72      * simplification is possible
73      */
74
75     if (tailAcceptingStatus[i] == 1) {
76
77       for (j=0; j<k; j++) {
78
79         /* This checks if the last state in loop j is final */
80         if (loopAcceptingStatus[j]
81                     [(trueLoopHead[j]+y[j]-1)%y[j]] == 1) {
82
83           /* Simplification can occur */
84           simplify = 1;
85           break;
86
87         } /*if*/
88
89       } /*for*/
90
91       if (simplify) {
92
93         /* Reduce tail length by 1 */
94         *trueTailLength -= 1;
95
96         /* Cycle the head of each loop back by one */
97         for (j=0; j<k; j++) {
98
99           trueLoopHead[j] = (trueLoopHead[j]+y[j]-1)%y[j];
100
101        } /*for*/
102
103      } /*if*/
```

```
104
105     } /*if*/
106
107     /*
108      * If the last state in the tail is not final and none of
109      * the last states in the k loops are final, then
110      * simplification is also possible
111      */
112
113     else {
114
115       simplify = 1;
116
117       for (j=0; j<k; j++) {
118
119         /* This checks if the last state in loop j is final */
120         if (loopAcceptingStatus[j]
121                     [(trueLoopHead[j]+y[j]-1)%y[j]] == 1) {
122
123           /* Simplification can not occur */
124           simplify = 0;
125           break;
126
127         } /*if*/
128
129       } /*for*/
130
131       if (simplify) {
132
133         /* Reduce tail length by 1 */
134         *trueTailLength -= 1;
135
136         /* Cycle the head of each loop back by one */
137         for (j=0; j<k; j++) {
138
139           trueLoopHead[j] = (trueLoopHead[j]+y[j]-1)%y[j];
140
141         } /*for*/
142
143       } /*if*/
144
145     } /*else*/
```

```
146
147     if (simplify == 0) break;
148
149   } /*for*/
150
151   return 0;
152
153 } /*ChrNFSimplification*/
154
155
156 /*
157  * NEW - 07 Jun 2002
158  *
159  * This procedure computes a regular expression r specifying the
160  * language accepted by the input unary NFA.
161  *
162  * This procedure uses Horner's Rule to output a short regular
163  *   expression.
164  *
165  * Input:
166  *   tailAcceptingStatus is an int array of size trueTailLength;
167  *     tailAcceptingStatus[i] equals 1 if and only if q_i is a
168  *     final state
169  *   trueTailLength is the length of the tail after
170  *     simplification; this value could be anywhere between
171  *     1 and m, inclusive
172  *   loopAcceptingStatus is an array of k pointers; the ith
173  *     pointer points to an int array of size y_i;
174  *     loopAcceptingStatus[i][j] equals 1 if and only if p_i,j
175  *     is a final state, where 0 <= j < y_i
176  *   trueLoopHead is an int array of size k; trueLoopHead[i]
177  *     denotes the index of the true head of loop i after
178  *     simplification; this value could be anywhere between 0
179  *     and y_i-1, inclusive
180  *
181  * Convention:  "" denotes the empty string and 0 denotes the
182  *   empty language.
183  */
184 int ComputeRegExp(int k, int *y,
185                   int *tailAcceptingStatus,
186                   int trueTailLength,
187                   int **loopAcceptingStatus,
```

```
188                 int *trueLoopHead) {
189
190   /*
191    * numParentheses is required to output the correct number of
192    *   closing parentheses at the end
193    * numInnerParentheses is required to output the correct
194    *   number of closing parentheses for the regexp specifying
195    *   each loop
196    * distanceToLastFinalState is the distance to the last final
197    *   state found -- this quantity is essential when using
198    *   Horner's rule
199    * tailNull is 1 if a final state has not been found in the
200    *   tail; once a final state has been found, it changes to 0
201    * loopsNull is 1 if a final state has not been found in ANY
202    *   loops; once a final state has been found, it changes to 0
203    * loopNull is 1 if a final state has not been found in a
204    *   particular loop; once a final state has been found, it
205    *   changes to 0
206    */
207
208   int i, j;
209   int flag;
210   int numParentheses = 0;
211   int numInnerParentheses = 0;
212   int distanceToLastFinalState = 0;
213   int tailNull, loopsNull, loopNull;
214
215   printf("The following regular expression generates the"
216          " language:\n");
217
218   /*
219    * The first thing we want to do is to check whether there are
220    * any final states at all.  If not, then L(M) is the empty
221    * language and we can quit immediately.
222    */
223
224   flag = 0;
225
226   for (i=0; i<trueTailLength; i++) {
227
228     if (tailAcceptingStatus[i] == 1) {
229
```

```
230       flag = 1;
231       break;
232
233     } /*if*/
234
235   } /*for*/
236
237   if (flag == 0) {
238
239     for (i=0; i<k; i++) {
240
241       for (j=0; j<y[i]; j++) {
242
243         if (loopAcceptingStatus[i][j] == 1) {
244
245           flag = 1;
246           break;
247
248         } /*if*/
249
250       } /*for*/
251
252       if (flag == 1) break;
253
254     } /*for*/
255
256   } /*if*/
257
258   if (flag == 0) {
259
260     /* L(M) is the empty language */
261     printf("0\n");
262     return 0;
263
264   } /*if*/
265
266
267   /* L(M) is not the empty language -- proceed */
268   tailNull = 1;
269
270   /* Compute a regular expression for the tail */
271   for (i=0; i<trueTailLength; i++) {
```

```
272
273     if (tailAcceptingStatus[i] == 1) {
274
275       if (distanceToLastFinalState == 0) {
276
277         /* q_0 is a final state */
278         printf("\"\"");
279
280       } /*if*/
281       else if (distanceToLastFinalState == 1) {
282
283         numParentheses += 1;
284
285         if (tailNull == 1) {
286
287           printf("a(\"\"");
288
289         } /*if*/
290         else {
291
292           printf("+a(\"\"");
293
294         } /*else*/
295
296       } /*else if*/
297       else {
298
299         numParentheses += 1;
300
301         if (tailNull == 1) {
302
303           printf("a^%d(\"\"", distanceToLastFinalState);
304
305         } /*if*/
306         else {
307
308           printf("+a^%d(\"\"", distanceToLastFinalState);
309
310         } /*else*/
311
312       } /*else*/
313
```

```
314       tailNull = 0;
315       distanceToLastFinalState = 0;
316
317     } /*if*/
318
319     distanceToLastFinalState += 1;
320
321   } /*for*/
322
323   /*
324    * If there are no loops, then we want to bypass this entire
325    * middle section and go right to the end.  Only enter the
326    * middle section if k is non-zero.
327    */
328
329   if (k != 0) {
330
331     /*
332      * Output the last portion of the regular expression before
333      * the innermost level of parentheses
334      */
335
336     if (distanceToLastFinalState == 1) {
337
338       if (tailNull) {
339
340         printf("a(");
341
342       } /*if*/
343       else {
344
345         printf("+a(");
346
347       } /*else*/
348
349     }
350     else {
351
352       if (tailNull) {
353
354         printf("a^%d(", distanceToLastFinalState);
355
```

```
356       } /*if*/
357       else {
358
359         printf("+a^%d(", distanceToLastFinalState);
360
361       } /*else*/
362
363     } /*else*/
364
365     numParentheses += 1;
366     loopsNull = 1;
367
368     /* Compute regular expressions for each of the loops */
369     for (i=0; i<k; i++) {
370
371       numInnerParentheses = 0;
372       distanceToLastFinalState = 0;
373       loopNull = 1;
374
375       /* Compute regular expression for loop i+1 */
376       for (j=0; j<y[i]; j++) {
377
378         if (loopAcceptingStatus[i]
379                     [(trueLoopHead[i]+j)%y[i]] == 1) {
380
381           /*
382            * Use loopNull and loopsNull to determine whether to
383            * output a leading + sign
384            */
385           if (loopsNull == 0 && loopNull == 1) {
386
387             printf("+");
388
389           } /*if*/
390
391           /* Output information */
392           if (distanceToLastFinalState == 0) {
393
394             /* p_i,0 is a final state */
395
396             /*
397              * NEW - 10 Jun 2002
```

```
398             * for simplification purposes, do not output
399             *   anything
400             */
401
402          } /*if*/
403          else if (distanceToLastFinalState == 1) {
404
405            /* Check loopNull for output format */
406            if (loopNull) {
407
408              printf("a");
409
410            } /*if*/
411            else {
412
413              printf("(\"\"+a");
414              numInnerParentheses += 1;
415
416            } /*else*/
417
418          } /*else if*/
419          else {
420
421            /* Check loopNull for output format */
422            if (loopNull) {
423
424              printf("a^%d", distanceToLastFinalState);
425
426            } /*if*/
427            else {
428
429              printf("(\"\"+a^%d", distanceToLastFinalState);
430              numInnerParentheses += 1;
431
432            } /*else*/
433
434          } /*else*/
435
436          loopNull = 0;
437          loopsNull = 0;
438          distanceToLastFinalState = 0;
439
```

```
440         } /*if*/
441
442         distanceToLastFinalState += 1;
443
444       } /*for*/
445
446       /*
447        * SPECIAL CASE - if loopNull equals 1, regular expression
448        *   specifying loop i is simply the empty regular
449        *   expression - we simply output nothing if this is the
450        *   case.  Otherwise, proceed as normal.
451        */
452
453       if (loopNull != 1) {
454
455         /*
456          * Output the correct number of parentheses for this
457          * subexpression
458          */
459
460         for (j=numInnerParentheses; j>0; j--) {
461
462           printf(")");
463
464         } /*for*/
465
466         /* Output the star portion */
467         if (y[i] == 1) {
468
469           printf("(a)*");
470
471         } /*if*/
472         else {
473
474           printf("(a^%d)*", y[i]);
475
476         } /*else*/
477
478       } /*if*/
479
480     } /*for*/
481
```

```
482   } /*if*/
483
484   /* Output the correct number of closing parentheses */
485   for (i=numParentheses; i>0; i--) {
486
487     printf(")");
488
489   } /*for*/
490
491   printf("\n");
492
493 } /*ComputeRegExp*/
494
```

regexp.h:

```
 1 #ifndef __REGEXP__
 2 #define __REGEXP__
 3
 4
 5
 6
 7 /*************************************************************
 8  * regexp operations
 9  *************************************************************/
10
11 /*
12  * This procedure accepts a unary NFA in ChrNF and simplifies
13  * it by eliminating as many states in the tail as possible.
14  */
15 int ChrNFSimplification(int m, int k, int *y,
16                         int *tailAcceptingStatus,
17                         int *trueTailLength,
18                         int **loopAcceptingStatus,
19                         int *trueLoopHead);
20
21 /*
22  * This procedure computes a regular expression r specifying
23  * the unary NFA described by the input parameters.
24  */
25 int ComputeRegExp(int k, int *y,
26                   int *tailAcceptingStatus,
```

```
27                  int trueTailLength,
28                  int **loopAcceptingStatus,
29                  int *trueLoopHead);
30
31 #endif
32
```

squarematrix.c:

```
 1 #include <stdlib.h>
 2 #include "squarematrix.h"
 3
 4
 5 /*************************************************************
 6  * squarematrix.c - Implementation of functions used to
 7  *                  manipulate the SQUAREMATRIX data type.
 8  *
 9  * 24 Sep 2001 - completed SquareMatrix functions
10  *************************************************************/
11
12
13 /*************************************************************
14  * SquareMatrix functions
15  *************************************************************/
16
17
18 /*
19  * Initializes a square matrix to size (dim * dim);
20  *   top left entry is M[0][0],
21  *   bottom right entry is M[dim-1][dim-1]
22  * All entries of M are initialized to 0
23  * Returns -1 if dim < 1
24  */
25 int SquareMatrixInitialize(SquareMatrix **M, int dim) {
26
27   int i, j;
28
29   if (dim < 1) return -1;
30
31   *M = (SquareMatrix *)malloc(dim*sizeof(int));
32   (*M)->data = (int **)malloc(dim*sizeof(int));
33
```

```
34   for (i=0; i<dim; i++) {
35
36     (*M)->data[i] = (int *)malloc(dim*sizeof(int));
37
38     for (j=0; j<dim; j++) {
39
40       (*M)->data[i][j] = 0;
41
42     } /*for*/
43
44   } /*for*/
45
46   (*M)->dimension = dim;
47   return 0;
48
49 } /*SquareMatrixInitialize*/
50
51
52 /*
53  * Deletes a square matrix
54  * Returns 0 if successful
55  */
56 int SquareMatrixNuke(SquareMatrix **M) {
57
58   int i;
59
60   for (i=0; i<(*M)->dimension; i++) {
61
62     free((*M)->data[i]);
63
64   } /*for*/
65
66   free((*M)->data);
67   free(*M);
68   return 0;
69
70 } /*SquareMatrixNuke*/
71
72
73 /*
74  * Sets M[row][col] := value
75  * Returns 0 if successful
```

```
 76  * Returns -1 if row or col is not in the range
 77  *   0..M->dimension-1
 78  */
 79 int UpdateEntryInSquareMatrix(SquareMatrix *M, int row,
 80                                int col, int value) {
 81
 82   if (row < 0 || col < 0
 83               || row >= M->dimension
 84               || col >= M->dimension) return -1;
 85
 86   M->data[row][col] = value;
 87   return 0;
 88
 89 } /*UpdateEntryInSquareMatrix*/
 90
 91
 92 /*
 93  * Sets value := M[row][col]
 94  * Returns 0 if successful
 95  * Returns -1 if row or col is not in the range
 96  *   0..M->dimension-1; in this event, value is unchanged
 97  */
 98 int GetEntryInSquareMatrix(SquareMatrix *M, int row,
 99                            int col, int *value) {
100
101   if (row < 0 || col < 0
102               || row >= M->dimension
103               || col >= M->dimension) return -1;
104
105   *value = M->data[row][col];
106   return 0;
107
108 } /*GetEntryInSquareMatrix*/
109
110
111 /*
112  * Returns the identity matrix of size dim
113  * Returns NULL if dim < 1
114  */
115 SquareMatrix * IdentityMatrix(int dim) {
116
117   int i;
```

```
118   SquareMatrix *NewMatrix;
119
120   if (dim < 1) return NULL;
121
122   SquareMatrixInitialize(&NewMatrix, dim);
123
124   for (i=0; i<dim; i++) {
125
126     UpdateEntryInSquareMatrix(NewMatrix, i, i, 1);
127
128   } /*for*/
129
130   return (SquareMatrix *)NewMatrix;
131
132 } /*IdentityMatrix*/
133
134
135 /*
136  * Sets B := A
137  * Returns 0 if successful
138  */
139 int SquareMatrixCopy(SquareMatrix *A, SquareMatrix **B) {
140
141   int i, j, value;
142   SquareMatrix *NewMatrix;
143
144   SquareMatrixInitialize(&NewMatrix, A->dimension);
145
146   /* Copy over the elements from A */
147   for (i=0; i<A->dimension; i++) {
148
149     for (j=0; j<A->dimension; j++) {
150
151       GetEntryInSquareMatrix(A, i, j, &value);
152       UpdateEntryInSquareMatrix(NewMatrix, i, j, value);
153
154     } /*for*/
155
156   } /*for*/
157
158   *B = NewMatrix;
159   return 0;
```

```
160
161 } /*SquareMatrixCopy*/
162
163
164 /*
165  * Sets C := A + B
166  * Returns 0 if successful
167  * Returns -1 if dimensions of A and B do not match
168  */
169 int Add(SquareMatrix *A, SquareMatrix *B, SquareMatrix **C) {
170
171   int i, j, a_val, b_val;
172   SquareMatrix *NewMatrix;
173
174   /* Check that the dimensions match */
175   if (A->dimension != B->dimension) return -1;
176
177   SquareMatrixInitialize(&NewMatrix, A->dimension);
178
179   /* Add entries one by one */
180   for (i=0; i<A->dimension; i++) {
181
182     for (j=0; j<A->dimension; j++) {
183
184       GetEntryInSquareMatrix(A, i, j, &a_val);
185       GetEntryInSquareMatrix(B, i, j, &b_val);
186       UpdateEntryInSquareMatrix(NewMatrix, i, j, a_val+b_val);
187
188     } /*for*/
189
190   } /*for*/
191
192   /* Set C := A + B */
193   *C = NewMatrix;
194   return 0;
195
196 } /*Add*/
197
198
199 /*
200  * Sets C := AB
201  * Returns 0 if successful
```

```
202  * Returns -1 if dimensions of A and B do not match
203  */
204 int Multiply(SquareMatrix *A,
205              SquareMatrix *B,
206              SquareMatrix **C) {
207
208   int i, j, k, temp_sum, a_val, b_val;
209   SquareMatrix *NewMatrix;
210
211   /* Check that the dimensions match */
212   if (A->dimension != B->dimension) return -1;
213
214   SquareMatrixInitialize(&NewMatrix, A->dimension);
215
216   /* Multiply - a lot trickier than adding */
217   for (i=0; i<A->dimension; i++) {
218
219     for (j=0; j<A->dimension; j++) {
220
221       temp_sum = 0;
222
223       for (k=0; k<A->dimension; k++) {
224
225         GetEntryInSquareMatrix(A, i, k, &a_val);
226         GetEntryInSquareMatrix(B, k, j, &b_val);
227         temp_sum += a_val*b_val;
228
229       } /*for*/
230
231       UpdateEntryInSquareMatrix(NewMatrix, i, j, temp_sum);
232
233     } /*for*/
234
235   } /*for*/
236
237   /* Set C := AB */
238   *C = NewMatrix;
239   return 0;
240
241 } /*Multiply*/
242
243
```

```
244 /*
245  * Sets B := A^n
246  * Returns 0 if successful
247  * Returns -1 if n < 0
248  * As usual, A^0 == I, where I is the identity matrix of the
249  *   same size as A
250  */
251 int Exponentiate(SquareMatrix *A, int n, SquareMatrix **B) {
252
253   int i, j;
254   SquareMatrix *NewMatrix, *TempMatrix1, *TempMatrix2;
255
256   /* Check that n is non-negative */
257   if (n < 0) return -1;
258
259   /* Take action depending on the value of n */
260   if (n == 0) {
261
262     /* Set B to be the identity matrix */
263     *B = IdentityMatrix(A->dimension);
264     return 0;
265
266   } /*if*/
267   else if (n == 1) {
268
269     /* Set B to be equal to A */
270     SquareMatrixCopy(A, B);
271     return 0;
272
273   } /*else if*/
274   else if (n%2 == 0) {
275
276     /* n is even - use fast exponentiation algorithm */
277     Exponentiate(A, n/2, &TempMatrix1);
278     Exponentiate(A, n/2, &TempMatrix2);
279     Multiply(TempMatrix1, TempMatrix2, &NewMatrix);
280
281   } /*else if*/
282   else {
283
284     /* n is odd - use fast exponentiation algorithm */
285     Exponentiate(A, n/2, &TempMatrix1);
```

```
286     Exponentiate(A, n/2+1, &TempMatrix2);
287     Multiply(TempMatrix1, TempMatrix2, &NewMatrix);
288
289   } /*else*/
290
291   *B = NewMatrix;
292   SquareMatrixNuke(&TempMatrix1);
293   SquareMatrixNuke(&TempMatrix2);
294   return 0;
295
296 } /*Exponentiate*/
297
```

squarematrix.h:

```
 1 #ifndef __SQUAREMATRIX__
 2 #define __SQUAREMATRIX__
 3
 4
 5
 6
 7 /************************************************************
 8  * SQUAREMATRIX data structure
 9  ************************************************************/
10
11 typedef struct SquareMatrix_t {
12   int dimension;
13   int **data;
14 } SquareMatrix;
15
16 /************************************************************
17  * SquareMatrix operations
18  ************************************************************/
19
20 /*
21  * Initializes a square matrix to size (dim * dim);
22  *   top left entry is M[0][0],
23  *   bottom right entry is M[dim-1][dim-1]
24  * All entries of M are initialized to 0
25  * Returns -1 if dim < 1
26  */
27 int SquareMatrixInitialize(SquareMatrix **M, int dim);
```

```
28
29 /*
30  * Deletes a square matrix
31  * Returns 0 if successful
32  */
33 int SquareMatrixNuke(SquareMatrix **M);
34
35 /*
36  * Sets M[row][col] := value
37  * Returns 0 if successful
38  * Returns -1 if row or col is not in the range
39  *   0..M->dimension-1
40  */
41 int UpdateEntryInSquareMatrix(SquareMatrix *M, int row,
42                               int col, int value);
43
44 /*
45  * Sets value := M[row][col]
46  * Returns 0 if successful
47  * Returns -1 if row or col is not in the range
48  *   0..M->dimension-1; in this event, value is unchanged
49  */
50 int GetEntryInSquareMatrix(SquareMatrix *M, int row,
51                            int col, int *value);
52
53 /*
54  * Returns the identity matrix of size dim
55  * Returns NULL if dim < 1
56  */
57 SquareMatrix * IdentityMatrix(int dim);
58
59 /*
60  * Sets B := A
61  * Returns 0 if successful
62  */
63 int SquareMatrixCopy(SquareMatrix *A, SquareMatrix **B);
64
65 /*
66  * Sets C := A + B
67  * Returns 0 if successful
68  * Returns -1 if dimensions of A and B do not match; in this
69  *   case, C is unchanged
```

```
 */
int Add(SquareMatrix *A, SquareMatrix *B, SquareMatrix **C);

/*
 * Sets C := AB
 * Returns 0 if successful
 * Returns -1 if dimensions of A and B do not match; in this
 *   case, C is unchanged
 */
int Multiply(SquareMatrix *A,
             SquareMatrix *B,
             SquareMatrix **C);

/*
 * Sets B := A^n
 * Returns 0 if successful
 * Returns -1 if n < 0; in this case, B is unchanged
 * As usual, A^0 == I, where I is the identity matrix of the
 *   same size as A
 */
int Exponentiate(SquareMatrix *A, int n, SquareMatrix **B);

#endif
```

unaryNFA.c:

```
#include <stdlib.h>
#include "list.h"
#include "graph.h"
#include "unaryNFA.h"
#include "chrnf.h"


/*************************************************************
 * unaryNFA.c - Implementation of functions used to manipulate
 *              the NFA data type.
 *
 * 26 Sep 2001 - in progress
 * 10 Oct 2001 - completed StronglyConnectedComponents
 * 12 Oct 2001 - modified IsAccepted to take sccID into account
 * 17 Oct 2001 - changed IsAccepted to FindAcceptingStates
```

```
16  *               It now returns ALL accepting states in M_new
17  *               in one call, instead of returning a Y/N for a
18  *               particular input; this will reduce the running
19  *               time by O(n)
20  * 19 Oct 2001 - found a bug in FindAcceptingStates.
21  *               Essentially, the problem is that I was using an
22  *               int array, statesNewSCC, to keep track of the
23  *               strongly connected component which is the
24  *               closest predecessor to a given state.
25  *               However, this SCC need not be unique - a state
26  *               can have many SCCs which are closest.  Thus, I
27  *               replaced the int array with a List to allow for
28  *               this situation.  Also, I added some
29  *               modularization of the code
30  * 06 Apr 2002 - made a slight modification in
31  *               FindAcceptingStates - do not insert the sccID
32  *               of q_0 to the SCC lists if q_0 does not belong
33  *               to any SCC
34  * 07 Jun 2002 - moved output statements to chrnf.c, now return
35  *               final state information to chrnf.c via the two
36  *               arrays tailAcceptingStatus and
37  *               loopAcceptingStatus
38  *
39  ************************************************************/
40
41
42 /************************************************************
43  * unaryNFA functions
44  ************************************************************/
45
46
47 /*
48  * PRIVATE - copies data from NewArray to OldArray and
49  * from NewListArray to OldListArray
50  * Also, sets NewArray to the zero array and sets NewListArray
51  * to the empty list
52  * size is the size of OldArray and NewArray
53  * Returns 0 if successful
54  */
55 int CopyDataFromNewToOld(int *OldArray,
56                          int *NewArray,
57                          int size,
```

```
58                          List **OldListArray,
59                          List **NewListArray) {
60
61   int i, firstItem, secondItem;
62
63   for (i=0; i<size; i++) {
64
65     /* Copy element i from NewArray to OldArray */
66     OldArray[i] = NewArray[i];
67     NewArray[i] = 0;
68
69     /* Make OldListArray[i] the empty list */
70     while ((ReturnLastElemInList(OldListArray[i],
71             &firstItem, &secondItem) != -1));
72
73     /*
74      * Move elements of NewListArray[i] over to OldListArray[i]
75      * After this while loop, NewListArray[i] will be empty
76      */
77     while ((ReturnLastElemInList(NewListArray[i],
78             &firstItem, &secondItem) != -1)) {
79
80       ListInsert(OldListArray[i], firstItem, secondItem);
81
82     } /*while*/
83
84   } /*for*/
85
86   return 0;
87
88 } /*CopyDataFromNewToOld*/
89
90
91 /*
92  * PRIVATE - updates strongly connected component (SCC) info
93  * sccID is the strongly connected component vertex toVertex
94  * belongs to
95  */
96 int UpdateSCCInfo(List **OldListArray, List **NewListArray,
97                   int sccID, int fromVertex, int toVertex) {
98
99   int firstItem, secondItem;
```

```
100   List *tempList;
101
102   ListInitialize(&tempList, SINGLE);
103
104   /*
105    * Update SCC info here - if current vertex is in a SCC, then
106    * clear the existing list and use the new sccID.
107    * Otherwise, current vertex is not in any SCC:  in this case,
108    * add the old sccID to the list
109    */
110   if (sccID != -1) {
111
112     while (ReturnLastElemInList(NewListArray[toVertex],
113                                 &firstItem, &secondItem) != -1);
114
115     ListInsert(NewListArray[toVertex], sccID, 0);
116
117   } /*if*/
118   else {
119
120     /* Copy OldListArray[fromVertex] to tempList */
121     while (ReturnLastElemInList(OldListArray[fromVertex],
122                                 &firstItem,
123                                 &secondItem) != -1) {
124       ListInsert(tempList, firstItem, secondItem);
125
126     } /*while*/
127
128     /* Restore OldListArray[fromVertex] */
129     while (ReturnLastElemInList(tempList,
130                                 &firstItem,
131                                 &secondItem) != -1) {
132
133       ListInsert(OldListArray[fromVertex],
134                  firstItem, secondItem);
135
136       /*
137        * Insert items into NewListArray[toVertex], but avoid
138        * duplicates
139        */
140
141       if (!isElemInList(NewListArray[toVertex],
```

```
142                         firstItem,
143                         secondItem)) {
144
145         ListInsert(NewListArray[toVertex],
146                    firstItem, secondItem);
147
148       } /*if*/
149
150     } /*while*/
151
152   } /*else*/
153
154   ListNuke(&tempList);
155
156 } /*UpdateSCCInfo*/
157
158
159 /*
160  * PRIVATE - determines if state value is out of range
161  * Returns 1 if true, 0 if false
162  */
163 int StateValueOutOfRange(unaryNFA *M, int q) {
164
165   if (q < 0 || q >= M->numStates) return 1;
166   return 0;
167
168 } /*StateValueOutOfRange*/
169
170
171 /*
172  * PRIVATE - finds the maximum element in Array and returns it
173  * Array has size elements, all non-negative
174  * If size==0, return 0
175  */
176 int FindMax(int *Array, int size) {
177
178   int currentMax = 0, index;
179
180   for (index=0; index<size; index++) {
181
182     if (Array[index] > currentMax) {
183
```

```
184       currentMax = Array[index];
185
186     } /*if*/
187
188   } /*for*/
189
190   return currentMax;
191
192 } /*FindMax*/
193
194
195 /*
196  * Initializes a unary NFA
197  * M starts with no transitions and no final states
198  * Returns 0 if successful
199  */
200 int unaryNFAInitialize(unaryNFA **M, int numStates) {
201
202   int i;
203
204   *M = (unaryNFA *)malloc(sizeof(unaryNFA));
205   GraphInitialize(&(*M)->transitions, numStates);
206   (*M)->numStates = numStates;
207   (*M)->finalStates = (int *)malloc(numStates*sizeof(int));
208
209   /* Initialize all states to be non-accepting states */
210   for (i=0; i<numStates; i++) {
211
212     (*M)->finalStates[i] = 0;
213
214   } /*for*/
215
216   return 0;
217
218 } /*unaryNFAInitialize*/
219
220
221 /*
222  * Deletes a unary NFA
223  */
224 int unaryNFANuke(unaryNFA **M) {
225
```

```
226   GraphNuke(&(*M)->transitions);
227   free((*M)->finalStates);
228   free(*M);
229   return 0;
230
231 } /*unaryNFANuke*/
232
233
234 /*
235  * Returns 1 if there is a transition from state p to state q
236  * in M, 0 otherwise
237  * Returns -1 if state values are not in the range
238  * [0..numStates-1]
239  */
240 int TransitionExists(unaryNFA *M, int p, int q) {
241
242   return IsAdjacent(M->transitions, p, q);
243
244 } /*TransitionExists*/
245
246
247 /*
248  * Adds a transition from state p to state q in unaryNFA M
249  * Returns 0 if successful
250  * Returns -1 if states p or q are not in the range
251  * [0..numStates-1]
252  */
253 int AddTransition(unaryNFA *M, int p, int q) {
254
255   return AddEdge(M->transitions, p, q);
256
257 } /*AddTransition*/
258
259
260 /*
261  * Deletes the transition from state p to state q in unaryNFA M
262  * Returns 0 if successful
263  * Returns -1 if states p or q are not in the range
264  * [0..numStates-1]
265  */
266 int DeleteTransition(unaryNFA *M, int p, int q) {
267
```

```
268   return DeleteEdge(M->transitions, p, q);
269
270 } /*DeleteTransition*/
271
272
273 /*
274  * Makes state q an accepting state
275  * Returns 0 if successful
276  * Returns -1 if state q is not in M
277  */
278 int MakeFinalState(unaryNFA *M, int q) {
279
280   if (StateValueOutOfRange(M, q)) return -1;
281
282   M->finalStates[q] = 1;
283   return 0;
284
285 } /*MakeFinalState*/
286
287
288 /*
289  * Makes state q a non-accepting state
290  * Returns 0 if successful
291  * Returns -1 if state q is not in M
292  */
293 int RemoveFinalState(unaryNFA *M, int q) {
294
295   if (StateValueOutOfRange(M, q)) return -1;
296
297   M->finalStates[q] = 0;
298   return 0;
299
300 } /*RemoveFinalState*/
301
302
303 /*
304  * Returns 1 if state q is a final state, 0 otherwise
305  * Returns -1 if state q is not in M
306  */
307 int IsFinalState(unaryNFA *M, int q) {
308
309   if (StateValueOutOfRange(M, q)) return -1;
```

```
310
311   return M->finalStates[q];
312
313 } /*IsFinalState*/
314
315
316 /*
317  * Finds and returns all accepting states in M', the unary NFA
318  * in CNF equivalent to M.
319  *
320  * First, this procedure finds all accepting states in the tail
321  * of M'.  Second, it finds all accepting states in the loops
322  * of M'.
323  *
324  * As usual:
325  *   m is the number of states in the tail of M',
326  *   k is the number of loops in M', and
327  *   y is an array denoting the lengths of the loops in M'.
328  *
329  * Input:
330  *   tailAcceptingStatus is an int array of size m,
331  *   loopAcceptingStatus is an array of k pointers; the ith
332  *     pointer points to an int array of size y_i
333  *
334  * Output:
335  *   tailAcceptingStatus[i] equals 1 if and only if q_i is a
336  *     final state
337  *   loopAcceptingStatus[i][j] equals 1 if and only if p_i,j is
338  *     a final state, where 0 <= j < y_i
339  */
340 int FindAcceptingStates(unaryNFA *M, int m, int k, int *y,
341       int *tailAcceptingStatus, int **loopAcceptingStatus) {
342
343   /*
344    * Idea:  Start by simulating the tail first, then simulate
345    *        the loops.  In the tail, simulate execution of M on
346    *        input a^(m-1).  Let n = number of states in M.
347    *        Use two arrays of size n to keep track of current
348    *        states.  After ith iteration, check if one of the
349    *        current states is accepting; if yes, then a^i is in
350    *        the language and output q_i as a final state.
351    *
```

```
352    *        In the loops, given an accepting path P in M, we
353    *        want to know which strongly connected component in
354    *        M was the last one traversed by P and make sure it
355    *        is equal to the corresponding loop before labeling a
356    *        state in M' as accepting - make use of the sccID to
357    *        accomplish this
358    */
359
360   int *prevStates, *currentStates;
361   List **statesOldSCC, **statesNewSCC;
362   int i, j, p, maxElem, firstItem, secondItem;
363
364   /* Allocate memory */
365   prevStates = (int *)malloc(M->numStates*sizeof(int));
366   currentStates = (int *)malloc(M->numStates*sizeof(int));
367
368   statesOldSCC = (List **)malloc(M->numStates*sizeof(int));
369   statesNewSCC = (List **)malloc(M->numStates*sizeof(int));
370
371   for (i=0; i<M->numStates; i++) {
372
373     ListInitialize(&statesOldSCC[i], SINGLE);
374     ListInitialize(&statesNewSCC[i], SINGLE);
375
376   } /*for*/
377
378   /* Initialize arrays */
379   for (i=0; i<m; i++) {
380
381     tailAcceptingStatus[i] = 0;
382
383   } /*for*/
384
385   for (i=0; i<k; i++) {
386
387     for (j=0; j<y[i]; j++) {
388
389       loopAcceptingStatus[i][j] = 0;
390
391     } /*for*/
392
393   } /*for*/
```

```
394
395   /* Start with the states in the tail first */
396   for (i=0; i<M->numStates; i++) {
397
398     prevStates[i] = 0;
399     currentStates[i] = 0;
400
401   } /*for*/
402
403   /* Begin with the start state */
404   currentStates[START_STATE] = 1;
405
406   /*
407    * NEW - 06 Apr 2002
408    * Do not insert if q_0 does not belong to any SCC
409    */
410   if (M->transitions->sccID[START_STATE] != -1) {
411     ListInsert(statesNewSCC[START_STATE],
412                M->transitions->sccID[START_STATE], 0);
413   } /*if*/
414
415   /* Now simulate execution of M on a^(m-1) */
416   for (i=0; i<m; i++) {
417
418     /*
419      * Check if any of the states in currentStates is a final
420      * state
421      */
422
423     for (j=0; j<M->numStates; j++) {
424
425       if (currentStates[j] == 1 && IsFinalState(M, j)) {
426
427         /* q_i is an accepting state */
428         tailAcceptingStatus[i] = 1;
429         break;
430
431       } /*if*/
432
433     } /*for*/
434
435     /* Copy info in current arrays over to old arrays */
```

```
436     CopyDataFromNewToOld(prevStates, currentStates,
437                          M->numStates, statesOldSCC,
438                          statesNewSCC);
439
440     /*
441      * Now, currentStates becomes the set of states reachable
442      * from prevStates in one step
443      */
444     for (j=0; j<M->numStates; j++) {
445
446       if (prevStates[j] == 1) {
447
448         for (p=0; p<M->numStates; p++) {
449
450           if (TransitionExists(M, j, p)) {
451
452             currentStates[p] = 1;
453
454             /* Update SCC info here */
455             UpdateSCCInfo(statesOldSCC, statesNewSCC,
456                           M->transitions->sccID[p], j, p);
457
458           } /*if*/
459
460         } /*for*/
461
462       } /*if*/
463
464     } /*for*/
465
466   } /*for*/
467
468   /* ASSERT:  i == m */
469
470   /*
471    * Now, handle states in the loops
472    * i denotes the cycle index
473    * Given i, j denotes the jth state in cycle i
474    *
475    * For j, loop enough times to cover the longest loop:
476    * maxElem = max{i=1..k}(y[i])
477    */
```

```
478   maxElem = FindMax(y, k);
479
480   for (j=0; j<maxElem; j++) {
481
482     for (i=0; i<k; i++) {
483
484       /* Can skip if j exceeds y[i] */
485       if (j >= y[i]) continue;
486
487       /*
488        * Check if any of the states in currentStates is a final
489        * state
490        */
491
492       for (p=0; p<M->numStates; p++) {
493
494         /*
495          * We have to make one extra check here - make sure the
496          * ID of the last strongly connected component traversed
497          * by the path from q_0 to p contains i
498          */
499         if (currentStates[p] == 1 &&
500             IsFinalState(M, p) &&
501             isElemInList(statesNewSCC[p], i, 0)) {
502
503           /* p_(i+1),j is an accepting state */
504           loopAcceptingStatus[i][j] = 1;
505           break;
506
507         } /*if*/
508
509       } /*for*/
510
511     } /*for*/
512
513     /* Copy info in current arrays over to old arrays */
514     CopyDataFromNewToOld(prevStates, currentStates,
515                          M->numStates, statesOldSCC,
516                          statesNewSCC);
517
518     /*
519      * Now, currentStates becomes the set of states reachable
```

```
520      * from prevStates in one step
521      */
522     for (i=0; i<M->numStates; i++) {
523
524       if (prevStates[i] == 1) {
525
526         for (p=0; p<M->numStates; p++) {
527
528           if (TransitionExists(M, i, p)) {
529
530             currentStates[p] = 1;
531
532             /* Update SCC info here */
533             UpdateSCCInfo(statesOldSCC, statesNewSCC,
534                           M->transitions->sccID[p], i, p);
535
536           } /*if*/
537
538         } /*for*/
539
540       } /*if*/
541
542     } /*for*/
543
544   } /*for*/
545
546   /* Free allocated memory */
547   free(prevStates);
548   free(currentStates);
549
550   for (i=0; i<M->numStates; i++) {
551
552     ListNuke(&statesOldSCC[i]);
553     ListNuke(&statesNewSCC[i]);
554
555   } /*for*/
556
557   free(statesOldSCC);
558   free(statesNewSCC);
559
560   return 0;
561
```

```
562 } /*FindAcceptingStates*/
563
564
565 /*
566  * Given a unary NFA M as input, returns the strongly connected
567  * components of M in T
568  * k is the number of strongly connected components of M
569  * Must allocate memory for T before calling
570  *
571  * Uses the algorithm in Introduction to Algorithms by Cormen,
572  * Leiserson and Rivest, Ch. 23, Sec. 5
573  */
574 int StronglyConnectedComponents(unaryNFA *M,
575                                 Graph **T,
576                                 int *k) {
577
578   int i;
579   Graph *H;
580
581   /* Call DFS to compute finishing times with flag == 0 */
582   DFS(M->transitions, NULL, 0, k);
583
584   /* Set H to be the transpose graph of M */
585   FindTranspose(M->transitions, &H);
586
587   /* Copy over finishing times to H */
588   for (i=0; i<H->numVertices; i++) {
589
590     H->finishingTime[i] = M->transitions->finishingTime[i];
591
592   } /*for*/
593
594   /*
595    * Call DFS on H to get strongly connected components with
596    * flag == 1
597    */
598
599   DFS(H, T, 1, k);
600
601   /* Copy over sccIDs from H to M */
602   for (i=0; i<H->numVertices; i++) {
603
```

```
604     M->transitions->sccID[i] = H->sccID[i];
605
606   } /*for*/
607
608   /* Don't need H anymore - delete it */
609   GraphNuke(&H);
610   return 0;
611
612 } /*StronglyConnectedComponents*/
613
```

unaryNFA.h:

```
 1 #ifndef __NFA__
 2 #define __NFA__
 3
 4 #include "list.h"
 5 #include "graph.h"
 6
 7 #define START_STATE 0
 8
 9
10
11 /*************************************************************
12  * unaryNFA data structure
13  *
14  * Assume Sigma = {a}
15  * States are represented by integers
16  * 0 denotes the start state
17  *************************************************************/
18
19 typedef struct unaryNFA_t {
20   Graph *transitions;
21   int numStates;
22   int *finalStates;
23 } unaryNFA;
24
25 /*************************************************************
26  * unaryNFA operations
27  *************************************************************/
28
29 /*
```

```
30  * Initializes a unary NFA
31  * M starts with no transitions and no final states
32  * Returns 0 if successful
33  */
34 int unaryNFAInitialize(unaryNFA **M, int numStates);
35
36 /*
37  * Deletes a unary NFA
38  */
39 int unaryNFANuke(unaryNFA **M);
40
41 /*
42  * Returns 1 if there is a transition from state p to state q in
43  *   M, 0 otherwise
44  * Returns -1 if state values are not in the range
45  *   [0..numStates-1]
46  */
47 int TransitionExists(unaryNFA *M, int p, int q);
48
49 /*
50  * Adds a transition from state p to state q in unaryNFA M
51  * Returns 0 if successful
52  * Returns -1 if states p or q are not in the range
53  *   [0..numStates-1]
54  */
55 int AddTransition(unaryNFA *M, int p, int q);
56
57 /*
58  * Deletes the transition from state p to state q in unaryNFA M
59  * Returns 0 if successful
60  * Returns -1 if states p or q are not in the range
61  *   [0..numStates-1]
62  */
63 int DeleteTransition(unaryNFA *M, int p, int q);
64
65 /*
66  * Makes state q an accepting state
67  * Returns 0 if successful
68  * Returns -1 if state q is not in M
69  */
70 int MakeFinalState(unaryNFA *M, int q);
71
```

```
 72 /*
 73  * Makes state q a non-accepting state
 74  * Returns 0 if successful
 75  * Returns -1 if state q is not in M
 76  */
 77 int RemoveFinalState(unaryNFA *M, int q);
 78
 79 /*
 80  * Returns 1 if state q is a final state, 0 otherwise
 81  * Returns -1 if state q is not in M
 82  */
 83 int IsFinalState(unaryNFA *M, int q);
 84
 85 /*
 86  * Finds all accepting states in M', the unary NFA in CNF
 87  * equivalent to M
 88  *
 89  * First, this procedure finds all accepting states in the tail
 90  *   of M'
 91  * Second, it finds all accepting states in the loops of M'.
 92  *
 93  * As usual:
 94  *   m is the number of states in the tail of M',
 95  *   k is the number of loops in M', and
 96  *   y is an array denoting the lengths of the loops in M'.
 97  *
 98  * Input:
 99  *   tailAcceptingStatus is an int array of size m
100  *   loopAcceptingStatus is an array of k pointers; the ith
101  *     pointer points to an int array of size y[i], 0 <= i < k
102  *
103  * Output:
104  *   tailAcceptingStatus[i] == 1 if q_i is a final state,
105  *     0 otherwise
106  *   loopAcceptingStatus[i-1][j] == 1 if p_i,j is a final state,
107  *     0 otherwise
108  */
109 int FindAcceptingStates(unaryNFA *M, int m, int k, int *y,
110       int *tailAcceptingStatus, int **loopAcceptingStatus);
111
112 /*
113  * Given a unary NFA M as input, returns the strongly connected
```

```
114  * components of M in T
115  * k is the number of strongly connected components of M
116  * Must allocate memory for T before calling
117  *
118  * Uses the algorithm in Introduction to Algorithms by Cormen,
119  * Leiserson and Rivest, Ch. 23, Sec. 5
120  */
121 int StronglyConnectedComponents(unaryNFA *M, Graph **T, int *k);
122
123
124 #endif
125
```

Bibliography

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[2] E. Bach and J. Shallit. *Algorithmic Number Theory*. The MIT Press, 1996.

[3] J. Berstel and L. Boasson. The set of minimal words of a context-free language is context-free. *Journal of Computer and System Sciences*, 55:477–488, 1997.

[4] A. Bertoni, M. Goldwurm, and N. Sabadini. The complexity of computing the number of strings of given length in context-free languages. *Theoretical Computer Science*, 86:325–342, 1991.

[5] A. Bertoni, M. Goldwurm, and M. Santini. Random generation for finitely ambiguous context-free languages. *Theoret. Informatics Appl.*, 35:499–512, 2001.

[6] N. Biggs. *Algebraic Graph Theory*. Cambridge University Press, 1974.

[7] J.-C. Birget. Partial orders on words, minimal elements of regular languages, and state complexity. *Theoretical Computer Science*, 119:267–291, 1993.

[8] M. Chrobak. Finite automata and unary languages. *Theoretical Computer Science*, 47:149–158, 1986.

[9] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* The MIT Press, 1990.

[10] Pál Dömösi. Unusual algorithms for lexicographical enumeration. *Acta Cybernet.*, 14:461–468, 2000.

[11] V. Gore, M. Jerrum, S. Kannan, Z. Sweedyk, and S. Mahaney. A quasi-polynomial time algorithm for sampling words from a context-free language. *Inform. and Comput.*, 134:59–74, 1997.

[12] T. Hickey and J. Cohen. Uniform random generation of strings in a context-free language. *SIAM J. Comput.*, 12:645–655, 1983.

[13] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[14] T. Jiang and B. Ravikumar. Minimal NFA problems are hard. *SIAM J. Comput.*, 22:1117–1141, 1993.

[15] T. Kasami. An efficient recognition and syntax algorithm for context-free languages. Scientific report afcrl-65-758, Air Force Cambridge Research Lab, Bedford, Mass., 1965.

[16] H. Leung. Separating exponentially ambiguous finite automata from polynomially ambiguous finite automata. *SIAM J. Comput.*, 27(4):1073–1082, 1998.

[17] B. Litow. Computing a context-free grammar-generating series. *Inform. and Comput.*, 169:174–185, 2001.

[18] H. Mairson. Generating words in a context-free language uniformly at random. *Inform. Process. Lett.*, 49:95–99, 1994.

[19] E. Mäkinen. On lexicographic enumeration of regular and context-free languages. *Acta Cybernet.*, 13:55–61, 1997.

[20] A. Mandel and I. Simon. On finite semigroups of matrices. *Theoretical Computer Science*, 5:101–111, 1977.

[21] C. Nicaud. Average state complexity of operations on unary automata. In *Mathematical Foundations of Computer Science*, pages 231–240, 1999.

[22] C. Papadimitriou. *Computational Complexity.* Addison-Wesley, 1994.

[23] B. Ravikumar and O. H. Ibarra. Relating the type of ambiguity of finite automata to the succinctness of their representation. *SIAM J. Comput.*, 18(6):1263–1282, 1989.

[24] D. Raymond and D. Wood. *Grail*: a C++ library for automata and expressions. *J. Symbolic. Comput.*, 17:341–350, 1994.

[25] J. Shallit. Numeration systems, linear recurrences and regular sets. *Inform. and Comput.*, 113:331–347, 1994.

[26] M. Sharir. A strong-connectivity algorithm and its application in data flow analysis. *Computers and Mathematics with Applications*, 7(1):67–72, 1981.

[27] R. Stearns and H. Hunt. On the equivalence and containment problems for unambiguous regular expressions, regular grammars and finite automata. *SIAM J. Comput.*, 14:598–611, 1985.

[28] A. Weber and H. Seidl. On the degree of ambiguity of finite automata. *Theoretical Computer Science*, 88:325–349, 1991.

[29] D. H. Younger. Recognition and parsing of context-free languages in time n^3. *Inform. and Control*, 10(2):189–208, 1967.

[30] S. Yu, Q. Zhuang, and K. Salomaa. The state complexities of some basic operations on regular languages. *Theoretical Computer Science*, 125:315–328, 1994.